NORTHWESTERN UNIVERSITY

Resource Management in Metacomputing Environments

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the Degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Warren Smith

EVANSTON, ILLINOIS

December 1999

ABSTRACT

Resource Management in Metacomputing Environments

Warren Smith

Metacomputing systems consist of various types of geographically distributed resources that users group into virtual computers, called metacomputers, to execute applications. In this work, we develop and evaluate general metacomputing services for resource selection and scheduling. To select which resources to use, users require information about resource performance, availability, and so on. We describe an information service that provides a common interface to such information.

One difficulty is that supercomputer schedulers do not typically provide information on when applications will execute. Therefore, we investigate techniques to predict these start times. We propose and evaluate a general technique for maintaining a historical database and using this database to predict characteristics of data points. We predict the execution time characteristic of applications and the wait time characteristic of scheduler states. We find that our run-time prediction errors are between 29 and 54 percent of the mean run times of the four workloads we use for evaluation and are significantly smaller than the errors of other run-time prediction techniques. If we use run-time predictions to predict wait times, our prediction error is 30 to 59 percent of the mean wait times. The disadvantage of this approach is that it requires detailed knowledge of the scheduling algorithm. If we use scheduler state as data points and construct wait-time predictions based on this, our prediction error is 49 to 94 percent of the mean wait times.

In addition to selecting which resources to use, users must be able to schedule access to the resources. To assist in this, we present a common interface to supercomputer scheduling systems. Further, many metacomputing applications require simultaneous access to resources and current scheduling systems do not provide this support when resources are controlled by more than one scheduler. To address this problem, we propose and evaluate techniques for reserving resources on supercomputers. We find that there are many different techniques to perform reservations with a range of effects on the wait times of queued jobs and the difference between when reservations are made and when they are initially requested.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in wide-area networking make it feasible to create applications that execute on virtual computers constructed from geographically distributed resources. An application may require a virtual computer consisting of several supercomputers, network connectivity between the computer systems, access to remote data sets, and connectivity to specialized resources such as scientific instruments and visualization devices. We call such virtual computers metacomputers and the applications that use them metacomputing applications. A set of software services such as authentication, communication, information retrieval, resource management, and access to remote data, are required to make efficient use of these resources. In this work, we develop and evaluate general metacomputing services for resource selection and scheduling.

To select resources for use by an application, a user requires information about resource configuration, performance, availability, and so on. In this work, we present an information service that provides a common interface to such information. A

good solution to this problem combines a resource representation that is precise enough to describe a wide range of resource characteristics, a query language that can express a wide range of resource requirements, and a search strategy that can match requirements with resources efficiently. Diverse resources such as workstations, SMPs, MPPs, networks, and scientific instruments must be represented.

One difficulty is that supercomputer schedulers do not typically provide information on when applications will execute. Therefore, we investigate techniques to predict these start times. We propose and evaluate a general technique for maintaining a historical database and using this database to predict characteristics of data points. We use this technique for predicting application execution times and for predicting queue wait times. When we predict application run times, the applications are described by the characteristics that a user provides when they submit an application to a parallel scheduler. We predict application run times using the run times of "similar" applications that have executed in the past. Initially, we do not know which characteristics to use to define which applications are similar and we do not know which of several statistical techniques to use to produce a prediction from similar applications. We search for the characteristics to use to define similar and how to produce a prediction using both greedy and genetic algorithm searches.

We use two techniques to predict queue wait times. The first technique uses run-time predictions and performs scheduling simulations of all the running and queued applications to predict when they will start executing. There are two disadvantages to this technique. First, exact knowledge of the scheduling algorithm is required and this knowledge can be difficult to determine about commercial scheduling systems. Second, this technique does not consider any applications that have not yet been submitted and with some scheduling algorithms, these applications can affect the

start times of applications that are already in queues. The second technique directly uses our prediction technique based on historical data. This technique characterizes the state of a scheduler and the application whose wait time is being predicted, finds similar scheduler/application states that have existed in the past, and then uses historical information of wait times in these similar states to produce a wait-time prediction. As previously stated, these wait-time predictions are useful when selecting which supercomputer to execute an application on. Further, if these predictions are accurate enough, they may also allow users to submit applications to several supercomputers so that they execute simultaneously and cooperate to solve a problem.

In addition to selecting which resources to use, users must be able to schedule access to the resources. To assist in this, we helped to define a common interface to supercomputer scheduling systems. This interface allows a user to start, monitor, and terminate applications and can be layered atop various scheduling systems or run on computer systems without schedulers. Further, many metacomputing applications require simultaneous access to resources and current scheduling systems do not provide this support when resources are controlled by more than one scheduler. To address this problem, we propose and evaluate techniques for reserving resources on supercomputers. We examine several different techniques for this and evaluate their performance based on changes in mean queue wait times and how near reservations are made to when the users initially request them.

As a prelude to our work on techniques for reserving supercomputing resources, we improve the performance of scheduling algorithms by using more accurate run-time predictions. Several scheduling algorithms use run-time predictions and typically use the maximum run-times that are specified by the users. We investigate

the effects on scheduling performance of using other run-time predictions besides maximum run times.

1.1 Contributions

This thesis provides the following contributions in the area of resource selection and scheduling in metacomputing environments:

- The design of an information service that allows users to locate resources suitable for their applications.
- The definition of a general prediction technique.
- An evaluation of our prediction technique when applied to predictions of the execution times of applications submitted to three different parallel computers.
- Two techniques for predicting when supercomputer schedulers will assign resources to applications.
- The design of a common interface to scheduling systems.
- An analysis of improvements to scheduling performance when more accurate run-time predictions are used.
- A definition and evaluation of a range of techniques for supporting advanced reservations in scheduling systems.

1.2 Outline

The next chapter describes metacomputing software with an emphasis on the Globus project, which uses some of the work developed here. Chapter 3 describes our general prediction technique, describes how we use this technique to predict application run times, presents performace data, and compares the performance of our technique to other run-time prediction techniques. Chapter 4 describes the two queue wait-time prediction techniques that we use to predict when schedulers will assign resources to applications and presents their performance. Chapter 5 evaluates the performance of several scheduling algorithms when our run-time predictions are used instead of other run-time predictions. This work is the prelude to the work in Chapter 6 where we modify the scheduling algorithms presented in Chapter 5 to support advance reservations of resources and discuss the performance of several different ways to support reservations. Chapter 7 describes the information service that we helped to design and Chapter 8 provides details of the common interface to scheduling systems that we helped to design. Finally, Chapter 9 presents our conclusions.

Chapter 2

Metacomputing

A major area of research activity in high-performance computing is to provide software so that distributed computers, databases, instruments, visualization devices, and other resources can be used together to solve scientific problems. This allows scientists to solve new and more complex problems than they could previously. Some application classes that are supported are:

1. *Desktop supercomputing.* These applications begin to execute on a user's desktop computer system and then acquire remote high-performance resources when needed. This gives the user the illusion that their desktop system is a supercomputer.

2. *Smart Instruments.* These applications connect instruments to computing resources and allow users to control the instruments and analyze their results on-line using the compute resources [58, 43].

3. *Collaborative environments.* These applications couple multiple virtual environments so that users in different locations can interact with each other and with supercomputer simulations [58].

4. *Distributed supercomputing.* These applications use multiple computers to simulate problems that are too large for a single computer or that can benefit from executing on different computer architectures [3, 9, 8].

In addition to the resources needed by the application classes described above, software is needed to manage the resources and allow applications to use them to solve their problems. These resources and the software to access them are called metacomputers [11] or computational grids [30]. Services for security, communication, resource management, information, access to remote data, and others are required and our provided by several software packages. The next section describes one such package, Globus, which uses some of the work presented here and the related work section will describe other sets of software for metacomputing.

Figure 2.1 shows an example of a user forming a metacomputer for use by an application. It illustrates several requirements that motivate our work. First, the user communicates with an information service to find suitable resources for the application. The user could have requirements for the architecture and/or operating system of the machine, speed of the machine, free disk space on the machine, and so on. Then, the user reduces the list of suitable supercomputers using wait-time predictions for various numbers of nodes and execution times. Further, if the schedulers support advanced reservation of resources for applications, the user can ask when various numbers of nodes can be reserved for sufficient amounts of time. In the example, we assume that the user determines that it would be better to execute

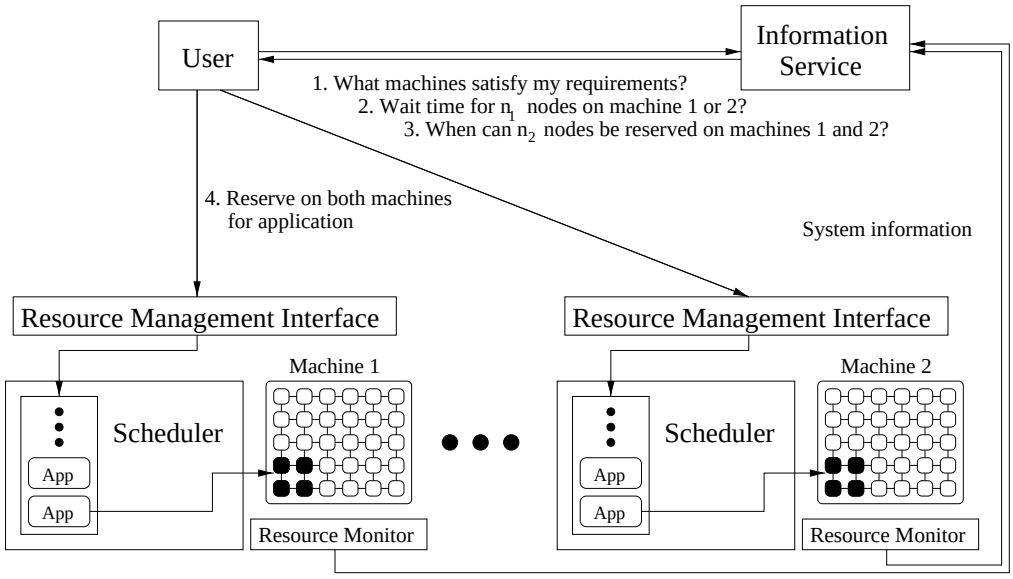

Figure 2.1: An example of forming a metacomputer.

the application using resources from two supercomputers. The user then uses a resource management interface to submit a reservation (or submit to the queue if they choose not to reserve resources) for the application to both scheduling systems and to monitor the application as it executes, even if the scheduling systems differ.

This example illustrates several software components that are very useful when forming metacomputers. An information service is helpful for finding appropriate resources to use and can serve as a virtually centralized information repository. Information on how long applications will wait before receiving resources from schedulers is useful for selecting which computer systems to use for an application. Advanced reservation of resources allows users to simultaneously allocate resources from more than one supercomputer for an application. Finally, a common interface to scheduling systems makes it much easier for a user to submit and monitor applications to many different scheduling systems.

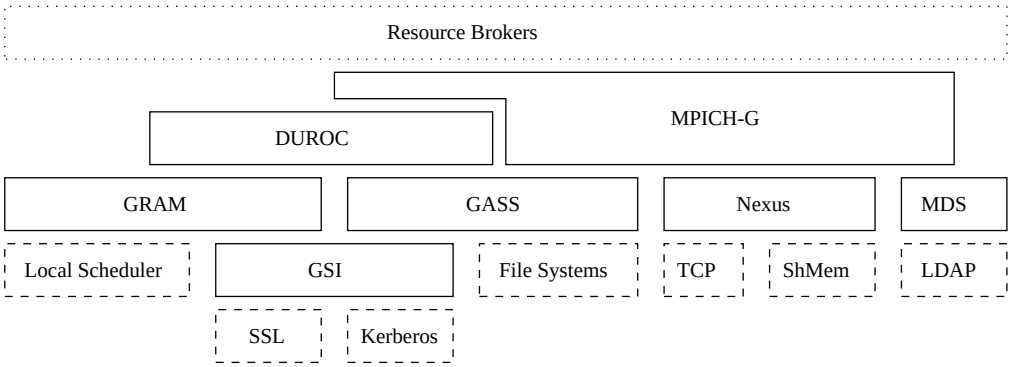

Figure 2.2: The Globus layered software architecture. The components with dashed borders are existing software. The components with solid borders are provided by Globus. The components with dotted lines are constructed by various groups.

The previous example concentrated on the metacomputing software components that we are concerned with in this work. There are many other components that assist in forming metacomputers. In the next section, we describe the Globus toolkit that provides software for using metacomputing systems and uses the information service and scheduling service described here. Section 2.2 describes other software to support metacomputing applications.

2.1 Globus Overview

Globus [29] consists of a layered software architecture that provides middleware software for metacomputing. This architecture is shown in Figure 2.2 and consists of the following components:

- *Nexus.* Nexus [27] is a library that provides threading, synchronization operations, and asynchronous remote procedure calls. A significant feature of Nexus is that it can be layered atop many different communication protocols (TCP

and shared memory are two examples). Nexus will select the best protocol to use to communicate between any pair of nodes [23]. For example, if an application uses two shared memory multiprocessors, RPCs between processes on the same system will be accomplished using shared memory, but RPCs between processes on different systems will be accomplished using TCP.

- *GSI.* The Globus Security Infrastructure (GSI) [26] provides authentication services using the Generic Security Services API (GSS-API) implemented over the Secure Socket Layer (SSL) public key infrastructure. The GSI also supports conversion between SSL certificates to Kerberos tickets. This allows authentication to Kerberos domains and services such as AFS.

- *MDS.* The Metacomputing Directory Service (MDS) [22] is an information service that we helped to design which provides information about computers, networks, instruments, applications, people, etc. to Globus users. The MDS is accessed using the Lightweight Directory Access Protocol (LDAP) [40], a widely-adopted protocol for accessing directory services.

- *GRAM.* The Globus Resource Allocation Manager (GRAM) [13] provides a common interface to local schedulers. We helped to define this interface which supports submission of applications, updates of the execution state of applications, and cancelling of applications. The GRAM consists of servers that execute on the computer systems providing resources and clients that a user uses to securely perform GRAM operations on servers.

- *DUROC.* The Dynamically Updatable Reconfigurable Online Coallocator (DUROC) [13] is used to execute applications on resources controlled by more than one GRAM. The DUROC allows a user to submit several GRAM jobs,

monitor their states, submit further GRAM jobs or remove already submitted GRAM jobs until a satisfactory resource pool is allocated. Then, the user releases a barrier which allows the application to begin executing and the user can monitor the states of the individual GRAM jobs and cancel the DUROC job. In addition to these control functions, the DUROC provides a runtime library that is compiled with the user's application. The user's application also must contain the barrier operation before it begins to compute.

- *MPICH-G.* MPICH is a portable implementation of the MPI [37] message passing protocol. This implementation contains low-level abstract devices that are used for communication, allowing MPICH to be implemented over many different communication protocols (for example, TCP, shared memory, and MPL). MPICH-G [24] is MPICH using the Globus device. The Globus device uses Nexus for communication, linking with the Globus libraries with mpicc, and providing support for using Globus mechanisms for job startup and cancel (GRAM and DUROC) inside of mpirun.

- *GASS.* The Globus Access to Secondary Storage (GASS) [5] provides mechanisms for transferring files to where they are needed. There are several different parts of GASS. First there is a GASS server that executes at a site and provides access to a user's files at that site. This server (or a FTP or HTTP server) is needed at remote sites to access files. There is a C API that provides open and close operations. These operations automatically move a remote file to a local cache when an open is performed and automatically moves the file back to the remote site (if necessary) when the file is closed. There is also a C API and programs to move files to and from remote systems.

- *Resource Brokers.* Resource brokers are higher-level tools that use the Globus middleware services. These brokers can be customized for different applications or types of applications. Globus is designed to support a wide variety of resource brokers but there are some local services that are not supported that make this difficult. These missing services motivate the rest of this work.

2.2 Related Work

There are other groups that provide metacomputing software and in this section, we describe many of them and compare them to the Globus toolkit. There are several systems that have been designed for metacomputing (Legin and Prospero) and several that have been extended to support metacomputing.

Legion [34, 36, 35] is a system that provides object-oriented distributed supercomputing. Legion supports object-oriented parallel programming with the Mentat programming language, an extension of C++, where objects may have their own address space. These objects can then be placed on remote systems with method calls to the objects transformed into messages between computer systems. Legion provides an object management system (OMS) that places objects on suitable computational nodes and a common file system, in the form of a persistent object space, where files are objects that live on disk.

The OMS places objects on nodes using the required architectures for the objects and information about the computer systems that are part of Legion. Candidate computer systems are determined by their architecture, FLOPS rating, MIPS rating, number of processors, load, number of objects, and cluster. A cluster is a set of systems that are close together, for example on the same subnet. The OMS does

not consider computer systems with too high of a load or too many objects residing there. The OMS will first try to place a new object in the same cluster as where it was created. One of the candidate computer systems is chosen using either a random or round-robin method. The OMS will try other clusters until a suitable system is found or until a maximum number of attempts are made.

Legion now provides many of the same services that Globus does. One main difference is that Legion is target to an object model of distributed computation, while Globus is not. Another main difference is in design philosophy. Legion is an integrated set of software while Globus is a set of software components that can be used together or separately. Another difference is that Legion has just added support for interfacing to local scheduling systems. Because this support is new, only a few scheduling systems are supported, and only one queue is used for each scheduler.

The **Prospero Resource Manager** (PRM) [50] is a tool for allocating processing resources to jobs and managing processing resources within each job. PRM has three main components: the system manager, the job manager, and the node manager. The system manager is responsible for a collection of nodes and maintains information about each node such as the architecture, the load, if the node is available and what job it is assigned to. The system manager responds to status updates from node managers and node requests from job managers. When a system manager receives a node request, it determines if the nodes are available and if so, allocates them to the job manager. If the nodes are not available, the system manager can allocate a subset of the nodes or reserve the nodes for when they become available. The node manager knows the architecture of the node on which it is executing and the load. It keeps its system manager informed of changes in load information and

the tasks that it is executing.

The job manager acts as an agent for a job and contacts system managers to find nodes for the job to execute on. The job manager locates system managers using a list provided by the user or the Prospero Directory Service (described below). The request language supported by the job manager allows the user to specify executables, arguments, number of nodes on which each executable should run, and the architectures on which the executable is available, among others. The job manager can locate system managers that have nodes of the desired architecture using a directory service and then sends requests to the system managers. If a job manager receives resources from a system manager, it assigns tasks of the job to nodes and contacts node managers to start the tasks. If a job manager does not receive resources from a system manager, it contacts other system managers. After the job starts, the job manager monitors the execution of the job, requesting additional nodes if necessary. Different job managers can be used depending on user requirements. For instance, a job manager that handles debugging tasks could be used.

Globus differs from the Prospero Resource Manager in several ways. First, Globus considers network and data resources along with compute resources when selecting resources for an application. Second, Globus provides more sophisticated mechanisms for using resources from multiple local schedulers (PRM system managers) in a single application. Third, Globus does not require that local schedulers be replaced as PRM does. Fourth, PRM does not coordinate the use of resources into the future, our resource managers perform this task. This may result in less efficient use of resources.

The **Prospero Directory Service** (Prospero) [48, 49] is a system to organize

information provided by many different sources such as Gopher, FTP, file systems, the World Wide Web, and others. Prospero is based on the Virtual System Model that organizes information as a global file system and allows customized views of the information. Organizations will have views of information that are organized in the most useful way to them: different organizations can have different views of the same information. Further, users have their own views of information so that they can easily locate information most useful to them.

Prospero does not provide mechanisms for searching for information. It does aid search tools by organizing information so that the scope of searches can be smaller. The Prospero Resource Manager can use the Directory Service to find system managers. This is accomplished by organizing a hierarchy directories where each directory corresponds to a set of characteristics of computational resources. The names of system managers can then be placed in this hierarchy wherever they provide resources of the type each directory corresponds to. PRM only uses the architecture and load of computational resources to characterize them so the hierarchy is limited.

In Globus, there are many more resource characteristics so the above method for locating resource providers is impractical. The MDS organizes information about more types of resources than Prospero and contains mechanisms to perform efficient searches on these resources. This is a necessary function when locating resources for an application in a large wide-area environment.

Condor [46, 7, 18, 45] is a distributed batch system that was designed to serve users who need more computational power than they have on their desk by allowing these users to run serial and parallel jobs on unutilized workstations. Condor consists of a central manager and several daemons running on all participating workstations (the workstation pool). The central manager keeps track of which machines are idle

and which machines have jobs they wish to run. When there are idle machines and machines with jobs to run, the central manager will pick machine a with jobs to run and an machine b which is idle. The central manager will inform machine a that it can run a job on machine b and inform machine b that machine a can run a job on it. Machine a will then start a job on machine b.

Condor determines if a machine is idle using the load of the machine and how long the keyboard and mouse have been unused. When a machine becomes idle, the central manager picks a machine to run a job on it. Which machine is picked is determined by the priority of each machine, how many jobs each machine wants to run, and how many jobs each machine is running. After a machine is picked, the job the machine sends off to run is determined using the priorities of the waiting jobs, when the job was submitted, and if the job has already completed some execution.

A recent addition to Condor is the ability to run jobs from one pool on workstations in another pool, if the pools trust each other. If pool d accepts jobs from pool c, then an idle machine in pool d advertises itself as an idle machine to pool c. If the central manager in pool c may then choose the machine from pool d on which to run jobs. The idle machine advertised from pool d is periodically changed.

Condor is also somewhat different from Globus. For example, Condor provides schedulers for each cluster of systems, while Globus does not provide a resource scheduler. Condor also does not interface to local scheduling systems, however, there is a GRAM that interfaces to Condor pools and work is ongoing on allowing Condor to send jobs to other systems by using Globus.

The **Load Sharing Facility** (LSF) [51, 53, 52] is a package sold by Platform Computing that was originally based on Condor. LSF consists of a central scheduler and daemons running on computational resources in the resource pool. LSF provides

mechanisms for several tasks such as a parallel make and a parallel shell. The core of the system schedules batch jobs to resources in the pool. Administrators can configure several queues to submit jobs to. The queues have properties such as when they are active, what resources they consider as candidates, the priority of the queue, the scheduling algorithm for jobs in the queue, and many others.

LSF has recently added a multi-cluster tool that allows a request submitted to one pool of resources to be executed on resources in another pool, similar to the Condor approach described above. This method does not allow a single application to use resources from multiple pools simultaneously. LSF differs from Globus in the same ways that Condor does.

The **AppLeS** (Application Level Scheduling) [4, 54] system acts as an agent for users. An AppLeS scheduler contains a model of how an application will perform on various sets of resources and it schedules the tasks that make up the application to compute resources using the performance of the application on the compute resources and considering the current and predicted future network performance between the resources. The AppLeS scheduler periodically reevaluates the resources the application is using and not using and may instruct the application to use different resources or use its current resources in a different way. AppLeS schedulers interact with resource management systems to find information about resources and acquire desired resources.

Globus differs in that it provides a resource management system that manages many resources and users but does not perform any application analysis. AppLeS will use Globus as one of the resource management systems that it interacts with.

Chapter 3

Run-Time Predictions

Predictions of application run time can be used to improve the performance of scheduling algorithms [31] and to predict how long a request at the head of a queue will wait for resources [16]. In later chapters, we show that run-time predictions can also be useful in high-performance distributed computing environments in two different ways. First, they are useful as a means of estimating queue times and hence guiding selections from among various resources. Second, they are useful when attempting to gain simultaneous access to resources from multiple scheduling systems [13].

The problem of predicting the run times of applications has been examined at many different levels. Several researchers [20, 38] have examined the execution characteristics of applications on parallel machines and analyzed this data for patterns. Others [54, 12] have performed detailed analysis of the execution of parallel programs for predicting application run times on different sets of compute and network resources. The work on run-time prediction that is most similar to that reported

here is performed by Downey [16] and Gibbons [31]. Both adopt the approach of making predictions for future applications by applying a "template" of application characteristics to identify "similar" applications that have executed in the past. Unfortunately, their techniques are not very accurate, with errors frequently exceeding execution times. The difference between these techniques and ours is that Downey and Gibbons both arbitrarily selected a set of templates to use, while we use search techniques to find the best set of templates.

Our approach to prediction is a general one and can be used to estimate one or more unknown characteristics of data points that are described by a set of characteristics. While this chapter presents our prediction technique in terms of predicting application execution times, in Chapter 4 we use this same approach to predict how long applications will wait until they receive resources. In that case, the data points are the characteristics of the scheduler state, the machine, and the application that is having it's wait time predicted. The characteristic being predicted is the queue wait time.

We believe that the key to making more accurate predictions is to be more careful about which past data points are used to make predictions. Accordingly, we apply greedy and genetic algorithm search techniques to identify templates that perform well when partitioning data points into categories within which data points are judged to be similar. We also examine and evaluate a number of variants of our basic prediction strategy. We examine whether it is useful to use regression techniques to exploit node count information when applications in a category have different node counts. We also examine the effect of varying the amount of past information used to make predictions, and we consider the impact of using user-supplied maximum run times on prediction accuracy.

We evaluate our techniques using four workloads recorded from supercomputer centers. This study shows that the use of search techniques makes a significant improvement in prediction accuracy: our prediction algorithm achieves prediction errors that are 21 to 61 percent lower than those achieved by Gibbons, depending on the workload, and 41 to 64 percent lower than those achieved by Downey. The templates found by the genetic algorithm search outperform the templates found by the greedy search.

The rest of this chapter is structured as follows. Section 3.1 describes how we define application similarity, perform predictions, and use search techniques to identify good templates. Section 3.2 describes the results when our algorithm is applied to supercomputer workloads. Section 3.3 compares our techniques and results with those of other researchers. Section 3.5 presents a summary of our results. Appendix A provides details of the statistical methods used in our work.

3.1 Prediction Techniques

Both intuition and previous work [16, 20, 31] indicate that "similar" applications are more likely to have similar run times than applications that have nothing in common. This observation is the basis for our approach to the prediction problem, which is to derive run-time predictions from historical information of previous similar runs.

To translate this general approach into a specific prediction method, we need to answer two questions:

1. *How do we define "similar"?* Jobs may be judged similar because they are submitted by the same user, at the same time, on the same computer, with the same arguments, on the same number of nodes, and so on. We require

techniques for answering the question: Are these two jobs similar?

2. *How do we generate predictions?* A definition of similarity allows us to partition a set of previously executed jobs into buckets or categories within which all are similar. We can then generate predictions by, for example, computing a simple mean of the run times in a category.

We structure the description of our approach in terms of these two issues.

3.1.1 Defining Similarity

In previous work, Downey [16] and Gibbons [31] demonstrated the value of using historical run-time information to identify "similar" jobs to predict run times for the purpose of improving scheduling performance and predicting wait times in queues. However, both Downey and Gibbons restricted themselves to relatively simple definitions of similarity. A major contribution of this work is to show that more sophisticated definitions of similarity can lead to significant improvements in prediction accuracy.

A difficulty in developing prediction techniques based on similarity is that two jobs can be compared in many ways. For example, we can compare the application name, submitting user name, executable arguments, submission time, and number of nodes requested. We can conceivably also consider more esoteric parameters such as home directory, files staged, executable size, and account to which the run is charged. We are restricted to those values recorded in workload traces obtained from various supercomputer centers. However, because the techniques that we propose are based on the automatic discovery of efficient similarity criteria, we believe that they will apply even if quite different information is available.

The workload traces that we consider are described in Table 3.1; they originate from Argonne National Laboratory (ANL), the Cornell Theory Center (CTC), and the San Diego Supercomputer Center (SDSC). Table 3.2 summarizes the information provided in these traces. Text in a field indicates that a particular trace contains the information in question; in the case of "Type," "Queue," or "Class" the text specifies the categories in question. The characteristics described in rows 1–9 are physical characteristics of the job itself. Characteristic 10, "maximum run time," is information provided by the user and is used by the ANL and CTC schedulers to improve scheduling performance. Rows 11 and 12 are temporal information, which we have not used in our work to date; we hope to evaluate the utility of this information in future work. Characteristic 13 is the run time that we seek to predict.

The general approach to defining similarity taken by ourselves, Downey, and Gibbons is to use characteristics such as those presented in Table 3.2 to define *templates* that identify a set of *categories* to which jobs can be assigned. For example, the template `(q,u)` specifies that jobs are to be partitioned by *queue* and *user*; on the SDSC Paragon, this template generates categories such as `(q16m,wsmith)`, `(q64l,wsmith)`, and `(q16m,foster)`.

We find that using discrete characteristics 1–8 in the manner just described works reasonably well. On the other hand, the number of nodes is an essentially continuous parameter, and so we prefer to introduce an additional parameter into our templates, namely, a "node range size" that defines what ranges of requested number of nodes are used to decide whether applications are similar. For example,

[1] Because of an error when the trace was recorded, the ANL trace does not include one-third of the requests actually made to the system. To compensate, we reduced the number of nodes on the machine from 120 to 80 when performing simulations.

Table 3.1: Characteristics of the workloads used in our studies.

Workload Name	System	Number of Nodes	Location	When	Number of Requests	Mean Run Time (minutes)
ANL [1]	IBM SP2	120	ANL	3 months of 1996	7994	97.40
CTC	IBM SP2	512	CTC	11 months of 1996	79302	182.18
SDSC95	Intel Paragon	400	SDSC	12 months of 1995	22885	107.76
SDSC96	Intel Paragon	400	SDSC	12 months of 1996	22337	166.48

Table 3.2: Characteristics recorded in workloads. The column "Abbr" indicates abbreviations used in subsequent discussion.

	Abbr	Characteristic	Argonne	Cornell	SDSC
1	t	Type	batch, interactive	serial, parallel, pvm3	
2	q	Queue			29 to 35 queues
3	c	Class		DSI/PIOFS	
4	u	User	Y	Y	Y
5	s	Loadleveler script		Y	
6	e	Executable	Y		
7	a	Arguments	Y		
8	na	Network adaptor		Y	
9	n	Number of nodes	Y	Y	Y
10		Maximum run time	Y	Y	
11		Submission time	Y	Y	Y
12		Start time	Y	Y	Y
13		Run time	Y	Y	Y

the template `(u, n=4)` specifies a node range size of 4 and generates categories `(wsmith, 1-4 nodes)` and `(wsmith, 5-8 nodes)`.

Once a set of templates has been defined (see Section 3.1.4), we can categorize a set of jobs (e.g., the workloads of Table 3.1) by assigning each job to those categories that match its characteristics. Categories need not be disjoint, and hence the same job can occur in several categories. If two jobs fall into the same category, they are judged similar; those that do not coincide in any category are judged dissimilar.

3.1.2 Generating Predictions

We now consider the question of how we generate run-time predictions. The input to this process is a set of templates T and a workload W for which run-time predictions are required. In addition to the characteristics described in the preceding section, a maximum history, type of data to store, and prediction type are also defined for each template. The maximum history indicates the maximum number of data points to store in each category generated from a template. The type of data is either an actual run time, denoted by `act`, or a relative run time, denoted by `rel`. A relative run time incorporates information about user-supplied run time estimates by storing the ratio of the actual run time to the user-supplied estimate (as described in Section 3.1.3). The prediction type determines how a run-time prediction is made from the data in each category generated from a template. We consider four prediction types: a mean, denoted by `mean`, a linear regression (`lin`), an inverse regression (`inv`), or a logarithmic regression (`log`).

The output from this process is a set of run-time predictions and associated confidence intervals. (As discussed in the appendix, a confidence interval is an interval centered on the run-time prediction within which the actual run time is expected

to appear some specified percentage of the time.) The basic algorithm is described below and comprises three phases: initialization, prediction, and incorporation of historical information.

1. Define T, the set of templates to be used, and initialize C, the (initially empty) set of categories.

2. At the time each application a begins to execute:

   (a) Apply the templates in T to the characteristics of a to identify the categories C_a into which the application may fall.

   (b) Eliminate from C_a all categories that are not in C or that cannot provide a valid prediction (i.e., do not have enough data points as described in the appendix).

   (c) For each category remaining in C_a, compute a run-time estimate and a confidence interval for the estimate.

   (d) If C_a is not empty, select the estimate with the smallest confidence interval as the run-time prediction for the application.

3. At the time each application a completes execution:

   (a) Identify the set C_a of categories into which the application falls. These categories may or may not exist in C.

   (b) For each category $c_i \in C_a$

      i. If $c_i \notin C$, create c_i in C.

      ii. If $|c_i| =$ maximum history(c_i), remove the oldest point in c_i.

      iii. Insert a into c_i.

Note that steps 2 and 3 operate asynchronously, since historical information for a job cannot be incorporated until the job finishes. Hence, our algorithm suffers from an initial ramp-up phase during which there is insufficient information in C to make predictions. This deficiency could be corrected by using a training set to initialize C.

We now discuss how a prediction is generated from the contents of a category in step 2(c) of our algorithm. We consider two techniques in this chapter. The first simply computes the mean of the run times contained in the category. The second attempts to exploit the additional information provided by the node counts associated with previous run times by performing regressions to compute coefficients a and b for the equations $R = aN + b$, $R = \frac{N}{a} + b$, and $R = a \log N + b$ for linear, inverse and logarithmic regressions, respectively. N is the number of nodes requested by the jobs, and R is the run time. The techniques used to compute confidence intervals for these predictors, are described in the appendix.

The use of maximum histories, referred to as `mh`, in step 3(b) of our algorithm allows us to control the amount of historical information used when making predictions and the amount of storage space needed to store historical information. A small maximum history means that less historical information is stored, and hence only more recent events are used to make predictions.

3.1.3 User Guidance

Another approach to obtaining accurate run-time predictions is to ask users to estimate the run time of an application at the time of submission. This approach may be viewed as complementary to the prediction techniques discussed previously, since historical information presumably can be used to evaluate the accuracy of user

predictions.

Unfortunately, none of the systems for which we have workload traces ask users to explicitly provide information about expected run times. However, all of the workloads provide implicit user estimates. The ANL and CTC workloads include user-supplied maximum run times. This information is interesting because users have some incentive to provide accurate estimates. The ANL and CTC systems both kill a job after its maximum run time has elapsed, so users have incentive not to underestimate this value. Both systems also use the maximum run time to determine when a job can be fit into a free slot, so users also have incentive not to overestimate this value.

Users also provide implicit estimates of run times in the SDSC workloads. The scheduler for the SDSC Paragon has many different queues with different priorities and different limits on application resource use. When users pick a queue to submit a request to, they implicitly provide a prediction of the resource use of their application. Queues that have lower resource limits tend to have higher priority, and applications in these queues tend to begin executing quickly; users are motivated to submit to queues with low resource limits. Also, the scheduler will kill applications that go over their resource limits, so users are motivated not to submit to queues with resource limits that are too low.

A simple approach to exploiting user guidance is to base predictions not on the run times of previous applications, but on the relationship between application run times and user predictions. For example, a prediction for the ratio of actual run time to user-predicted run time can be used along with the user-predicted run time of a particular application to predict the run time of the application. We use this technique for the ANL and CTC workloads by storing relative run times (i.e. the run

Table 3.3: Templates used by Gibbons for run-time prediction.

Number	Template	Predictor
1	(u,e,n,rtime)	mean
2	(u,e)	linear regression
3	(e,n,rtime)	mean
4	(e)	linear regression
5	(n,rtime)	mean
6	()	linear regression

times divided by the user-specified maximum run times) as data points in categories instead of the actual run times.

3.1.4 Template Definition and Search

We have not yet addressed the question of how we define an appropriate set of templates. This is a nontrivial problem. If too few categories are defined, we group too many unrelated jobs together and obtain poor predictions. On the other hand, if too many categories are defined, we have too few jobs in a category to make accurate predictions.

Downey and Gibbons both select a fixed set of templates to use for all of their predictions. Downey uses a single template containing only the queue name; prediction is based on a conditional probability function. Gibbons uses the six templates/predictor combinations listed in Table 3.3. The running time (`rtime`) characteristic indicates how long an application has been executing when a prediction is made for the application. Section 3.3 discusses further details of their approaches and presents a comparison with our work.

We use search techniques to identify good templates for a particular workload. While the number of application characteristics included in our traces is relatively

small, the fact that effective template sets may contain many templates means that an exhaustive search is impractical. Hence, we consider alternative search techniques. Results for greedy and genetic algorithm search are presented in this chapter.

The greedy and genetic algorithms both take as input a workload W from Table 3.1 and produce as output a template set; they differ in the techniques used to explore different template sets. Both algorithms evaluate the effectiveness of a template set T by applying the algorithm of Section 3.1.2 to workload W. Predicted and actual values are compared to determine for W and T both the mean error and the percentage of predictions that fall within the 90 percent confidence interval.

Greedy Algorithm

The greedy algorithm proceeds iteratively to construct a template set $T = \{t_i\}$ with each t_i of the form

$$\{\ ()\ (h_{1,1})\ (h_{2,1}, h_{2,2}), \ldots, (h_{i,1}, h_{i,2}, \ldots, h_{i,i})\ \},$$

where every $h_{j,k}$ is one of the n characteristics $h_1, h_2 \ldots, h_n$ from which templates can be constructed for the workload in question. The search over workload W is performed with the following algorithm:

1. Set the template set $T = \{()\}$.
2. For $i = 1$ to n
   (a) Set T_c to contain the $\binom{n}{i}$ different templates that contain i characteristics.
   (b) For each template t_c in T_c

i. Create a candidate template set $X_c = T \cup \{t_c\}$.

ii. Apply the algorithm of Section 3.1.2 to W and X_c, and determine mean error.

(c) Select the X_c with the lowest mean error, and add the associated template t_c to T.

Our greedy algorithm can search over any set of characteristics.

Genetic Algorithm

The second search algorithm that we consider uses genetic algorithm techniques to achieve a more detailed exploration of the search space. Genetic algorithms are a probabilistic technique for exploring large search spaces, in which the concept of cross-over from biology is used to improve efficiency relative to purely random search [33]. A genetic algorithm evolves individuals over a series of generations. The process for each generation consists of evaluating the fitness of each individual in the population, selecting which individuals will be mated to produce the next generation, mating the individuals, and mutating the resulting individuals to produce the next generation. The process then repeats until a stopping condition is met. The stopping condition we use is that a fixed number of generations have been processed. There are many different variations to this process, and we will next describe the variations we used.

Our individuals represent template sets. Each template set consists of between 1 and 10 templates, and we encode the following information in binary form for each template:

1. Whether a mean or one of the three regressions is used to produce a prediction

2. Whether absolute or relative run times are used

3. Whether each of the binary characteristics associated with the workload in question is enabled

4. Whether node information should be used and, if so, the range size from 1 to 512 in powers of 2

5. Whether the amount of history stored in each category should be limited and, if so, the limit between 2 and 65536 in powers of 2

The ranges for the number of nodes and the history are selected to be close to the maximum number of nodes requested by an application and the maximum number of applications in a workload that we are simulating.

A fitness function is used to compute the fitness of each individual and therefore its chance to reproduce. The fitness function should be selected so that the most desirable individuals have higher fitness and therefore have more offspring, but the diversity of the population must be maintained by not giving the best individuals overwhelming representation in succeeding generations. In our genetic algorithm, we wish to minimize the prediction error and maintain a range of individual fitnesses regardless of whether the range in errors is large or small. The fitness function we use to accomplish this goal is

$$F_{min} + \frac{E_{max} - E}{E_{max} - E_{min}} \times (F_{max} - F_{min}),$$

where E is the error of the individual (template set), E_{min} and E_{max} are the minimum and maximum errors of individuals in the generation, and F_{min} and F_{max} are the desired minimum and maximum fitnesses desired. We choose $F_{min} = 20$ and

$F_{max} = 80$ because for a set of experiments, these values are the best for finding the best individuals but maintaining the diversity of the population.

We use a common technique called stochastic sampling with replacement to select which individuals will mate to produce the next generation. In this technique, each parent is selected from the individuals by selecting individual i with probability $\frac{F_i}{\sum F}$.

The mating or crossover process is accomplished by randomly selecting pairs of individuals to mate and replacing each pair by their children in the new population. The crossover of two individuals proceeds in a slightly nonstandard way because our chromosomes are not fixed length but a multiple of the number of bits used to represent each template. Two children are produced from each crossover by randomly selecting a template i and a position p in the template from the first individual $T_1 = t_{1,1}, \ldots, t_{1,n}$ and randomly selecting a template j in the second individual $T_2 = t_{2,1}, \ldots, t_{2,m}$ so that the resulting individuals will not have more than 10 templates. The new individuals are then $\tilde{T}_1 = t_{1,1}, \ldots, t_{1,i-1}, n_1, t_{2,j+1}, \ldots, t_{2,m}$ and $\tilde{T}_2 = t_{2,1} \ldots t_{2,j-1}, n_2, t_{1,i+1}, \ldots, t_{i,n}$. If there are b bits used to represent each template, n_1 is the first p bits of $t_{1,i}$ concatenated with the last $b-p$ bits of $t_{2,j}$, and n_2 is the first p bits of $t_{2,j}$ concatenated with the last $b-p$ bits of $t_{1,i}$.

In addition to using crossover to produce the individuals of the next generation, we also use a process called elitism whereby the best individuals in each generation survive unmutated to the next generation. We use crossover to produce all but two individuals for each new generation and use elitism to select the last two individuals for each new generation. We choose two individuals using elitism because typically, only a very few individuals need to be choosen this way to improve performance and choosing two instead of one made the rest of the genetic algorithm easier to

implement. The individuals resulting from the crossover process are mutated to help maintain a diversity in the population. Each bit representing the individuals is flipped with a probability of 0.01.

3.2 Experimental Results

In the preceding section we described our basic approach to run-time prediction. We introduced the concept of *template search* as a means of identifying efficient criteria for selecting "similar" jobs in historical workloads. We also noted potential refinements to this basic technique, including the use of alternative search methods (greedy vs. genetic), the introduction of node count information via regression, support for user guidance, and the potential for varying the amount of historical information used. In the rest of this chapter, we discuss experimental studies that we have performed to evaluate the effectiveness of our techniques and the significance of the refinements just noted.

Our experiments used the workload traces summarized in Table 3.1 and are intended to answer the following questions:

- What is the relative effectiveness of the mean and regression predictors?
- What is the impact of user guidance as represented by the maximum run times provided on the ANL and CTC SPs?
- What is the impact of varying the number of nodes in each category on prediction performance?
- What is the impact of varying the maximum amount of history kept by each category?

- How effectively do our greedy and genetic search algorithms perform?
- What are the trends for the best templates in the workloads?
- How do our techniques compare with those of Downey and Gibbons?

Greedy Search

Figure 3.1 and Figure 3.2 shows the results of our first set of greedy searches for template sets. The characteristics searched over are the ones listed in Table 3.2. Actual run times are used as data points and a curve is shown for each of the predictors. Several trends can be observed from this data. First, adding a second template with a single characteristic results in the most dramatic improvement in performance. The addition of this template has the least effect for the CTC workload where performance is improved between 5 and 25 percent and has the greatest effect for the SDSC workloads which improve between 34 and 48 percent. The addition of templates using up to all possible characteristics results in less improvement than the addition of the template containing a single characteristic. The improvements range from 1 to 20 percent with the ANL workload seeing the most benefit and the SDSC96 workload seeing the least.

Second, the graphs show that for the final template set, the mean is a better predictor than any of the regressions. The final predictors obtained by using means are between 2 and 18 percent more accurate than those based on regressions. The impact of the choice of predictor on accuracy is greatest in the ANL workload and least in the CTC workload. If we search over the predictor as well as the other characteristics, the search results in a template set that is up to 11 percent better than any search using a particular predictor.

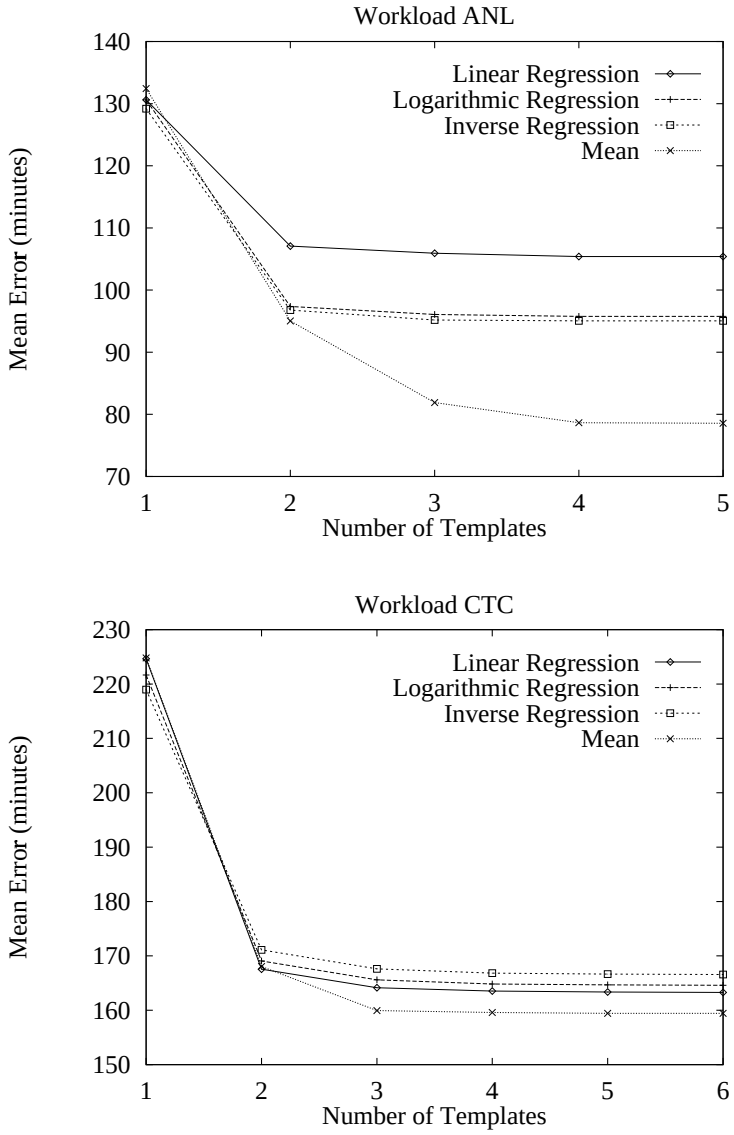

Figure 3.1: The mean errors of the greedy searches using run times as data points.

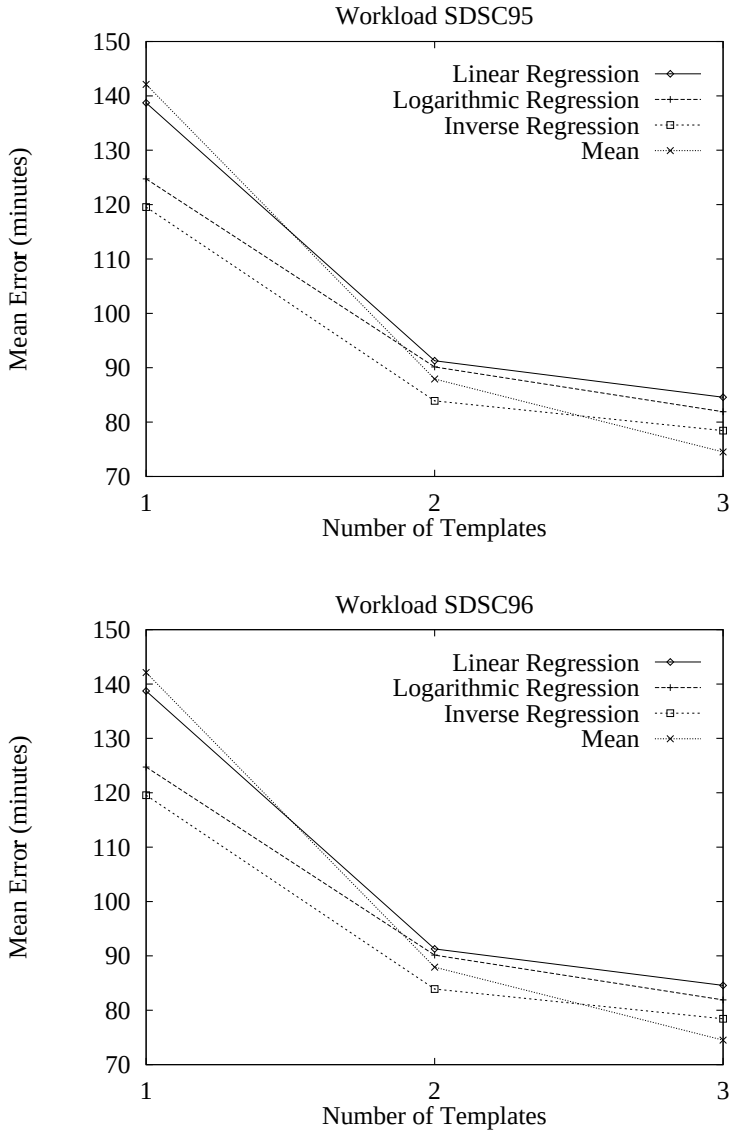

Figure 3.2: The mean errors of the greedy searches using run times as data points.

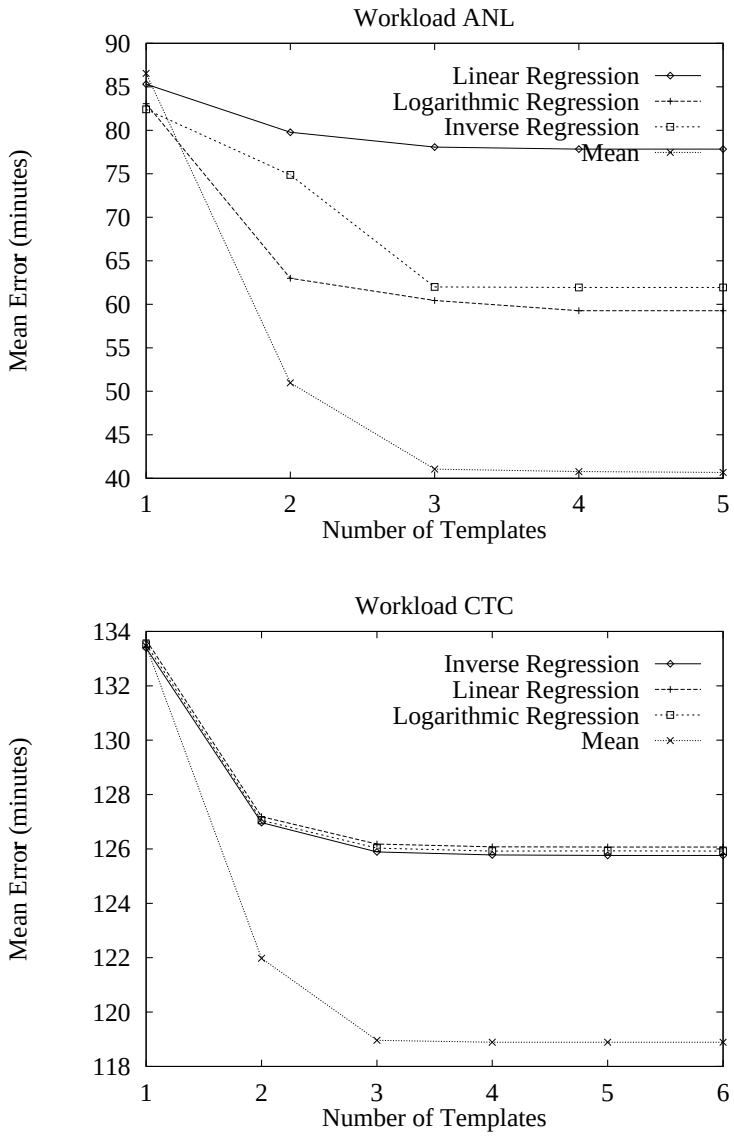

Figure 3.3: The mean errors of the greedy searches using relative run times as data points.

Figure 3.3 shows searches performed over the ANL and CTC workloads when relative run times are used as data points. Similar to the data in Figure 3.1, adding a second characteristic with a single characteristic results in the largest improvement in performance. Also, comparing the prediction errors in Figure 3.3 with the prediction errors in Figure 3.1 shows that using relative run times as data points results in a performance improvement of 19 to 48 percent.

Table 3.4 lists for each workload the accuracy of the best category templates found by the first set of greedy searches. In the last column, the mean error is expressed as a fraction of mean run time. Mean errors of between 42 and 70 percent of mean run times may appear high; however, as we will see later, these figures are comparable to those achieved by other techniques, and our subsequent searches perform significantly better.

Looking at the templates listed in Table 3.4, we observe that for the ANL and CTC workloads, the executable and user name are both important characteristics to use when deciding whether applications are similar. Examination of other data gathered during the experiments shows that these two characteristics are highly correlated: substituting the user name for executable or script name or vice versa in templates results in similar performance in many experiments. This observation may imply that users tend to run one application at a time on these parallel computers.

The templates selected for the SDSC workloads indicate that the user who submits an application is more important in determining application similarity than the queue to which an application is submitted. Furthermore, Figure 3.1 shows that adding the third template results in performance improvements of only 2 to 12 percent on the SDSC95 and SDSC96 workloads. Comparing this result with the greater improvements obtained when relative run times are used in the ANL and

Table 3.4: Best predictions found during greedy first search.

Workload	Predictor	Data Point	Template Set	Mean Error (minutes)	Percentage of Mean Run Time
ANL	mean	relative run time	`(), (e), (u,a), (t,u,a), (t,u,e,a)`	40.68	41.77
CTC	mean	relative run time	`(), (u), (u,s), (t,c,s), (t,u,s,ni), (t,c,u,s,ni)`	118.89	65.25
SDSC95	mean	run time	`(), (u), (q,u)`	75.56	70.12
SDSC96	mean	run time	`(), (u), (q,u)`	82.40	49.50

CTC workloads suggests that SDSC queue classes are not good user-specified run-time estimates. It would be interesting to use the resource limits associated with queues as maximum run times. However, this information is not contained in the trace data available to us.

We next performed a second series of greedy searches to identify the impact of using node information when defining categories. We used node ranges when defining categories as described in Section 3.1.1. The results of these searches in Table 3.5 show that using node information improves prediction performance by 1 to 9 percent for the best predictors, with the largest improvement for the San Diego workloads. This information and the fact that characteristics such as executable, user name, and arguments are selected before nodes when searching for templates indicates that the importance of node information to prediction accuracy is only moderate.

Further, the greedy search selects relatively small node range sizes coupled with user name or executable. This fact indicates, as expected, that an application executes for similar times on similar numbers of nodes.

Our third and final series of searches identify the impact of setting a maximum amount of history to use when making predictions from categories. The results of these searches are shown in Table 3.6 (because of time constraints, no results are available for the CTC workload). Comparing this data with the data in Table 3.5 shows that using a maximum amount of history improves prediction performance by 14 to 34 percent for the best predictors. The least improvement occurs for the ANL workload, the most for the SDSC96 workload. Other facts to note about the results in the table are that a maximum history is used in the majority of the templates and that when a maximum history is used, it is relatively small. The latter fact

Table 3.5: Best predictions found during second greedy search, using node information.

Workload	Predictor	Data Point	Template Set	Mean Error (minutes)	Percentage of Mean Run Time
ANL	mean	relative run time	`(), (e), (u,a), (t,u,n=2), (t,e,a,n=16), (t,u,e,a,n=32)`	39.87	40.93
CTC	mean	relative run time	`(), (u), (e,n=1), (t,u,n=1) (c,u,e,n=8), (t,c,u,ni,n=4) (t,c,u,e,ni,n=256)`	117.97	64.75
SDSC95	mean	run time	`(),(u),(u,n=1),(q,u,n=1)`	67.63	62.76
SDSC96	mean	run time	`(),(u),(u,n=4),(q,u,n=8)`	76.20	45.77

indicates that temporal locality is relevant in the workloads.

3.2.1 Genetic Algorithm Search

We now investigate the effectiveness of using a genetic algorithm to search for the best template sets by performing the same three series of searches. The results are shown in Table 3.7 through Table 3.9 along with the results of the corresponding greedy searches for comparison.

As shown in the tables, the best templates found during the genetic algorithm search provide mean errors that are 10 percent better to 1 percent worse than the best templates found during the greedy search. For the majority of the experiments, the genetic search outperforms the greedy search.

The best template sets identified by the genetic search procedure are listed in Table 3.10. This data shows that similar to the greedy searches, the mean is the single best predictor to use and using relative run times as data points, when available, provides the best performance.

Another observation is that node information and a maximum history are used throughout the best templates found during the genetic search. This confirms the observation made during the greedy search that using this information when defining templates results in improved prediction performance.

Figure 3.4 shows the progress of the two different search algorithms for a search of template sets for the ANL workload that use the number of nodes requested, limit the maximum history, either actual or relative run times, and use any of the predictors. The graph shows that while both searches result in template sets that have nearly the same accuracy, the genetic algorithm search does so much more quickly. This fact is important because a simulation that takes minutes or hours is

Table 3.6: Best predictions found during third greedy search.

Workload	Predictor	Data Point	Template Set	Mean Error (minutes)	Percentage of Mean Run Time
ANL	mean	relative run time	(),(e),(u,mh=4), (t,u,mh=16), (e,a,n=64,mh=4), (t,e,a,n=1,mh=8), (t,u,e,a,n=16,mh=8)	34.28	35.20
SDSC95	mean	run time	(),(u),(q,mh=4), (u,n=8,mh=8), (q,u,n=1,mh=32)	48.33	45.16
SDSC96	mean	run time	(),(u),(q,mh=4), (u,n=4,mh=4) (q,u,n=1,mh=16)	50.14	30.12

Table 3.7: Performance of the best templates found during first genetic algorithm search.

Workload	Genetic Algorithm		Greedy	
	Mean Error (minutes)	Percentage of Mean Run Time	Mean Error (minutes)	Percentage of Mean Run Time
ANL	39.32	40.37	40.68	41.77
CTC	107.02	58.74	118.89	65.25
SDSC95	65.27	60.57	75.56	70.12
SDSC96	80.37	48.28	82.40	49.50

Table 3.8: Performance of the best templates found during second genetic algorithm search.

Workload	Genetic Algorithm		Greedy	
	Mean Error (minutes)	Percentage of Mean Run Time	Mean Error (minutes)	Percentage of Mean Run Time
ANL	38.79	39.83	39.87	40.93
CTC	106.25	58.32	118.05	64.80
SDSC95	60.03	55.71	67.63	62.76
SDSC96	74.75	44.90	76.20	45.77

Table 3.9: Performance of the best templates found during third genetic algorithm search.

Workload	Genetic Algorithm		Greedy	
	Mean Error (minutes)	Percentage of Mean Run Time	Mean Error (minutes)	Percentage of Mean Run Time
ANL	34.52	35.44	34.28	35.20
CTC	98.28	53.95	no data	no data
SDSC95	43.20	40.09	48.33	45.16
SDSC96	47.47	28.51	50.14	30.12

Table 3.10: The best templates found during genetic algorithm search.

Workload	Predictor	Data Point	Template Set
ANL	mean	relative run time	(u,e,n=128,mh=16384), (u,e,n=16,mh=4), (t,e,n=16,mh=128), (t,u,n=4,mh=∞), (t,u,a,n=4,mh=4), (t,u,a,n=64,mh=4), (t,u,e,a,n=64,mh=32)
CTC	mean	relative run time	(u,n=64,mh=8), (c,s,n=32,mh=128), (c,s,n=32,mh=256), (c,u,ni,n=256,mh=128), (t,u,s,n=1,mh=16), (t,c,u,ni,n=256,mh=32), (t,c,u,ni,n=4,mh=16384), (t,c,u,s,ni,n=1,mh=4)
SDSC95	mean	run time	(q,u,n=4,mh=4), (q,u,n=4,mh=64), (q,u,n=4,mh=8)
SDSC96	mean	run time	(q,u,n=16,mh=1024), (q,u,n=2,mh=4), (q,u,n=4,mh=1024), (q,u,n=4,mh=2048), (q,u,n=4,mh=4096), (q,u,n=4,mh=65536), (q,u,n=4,mh=8)

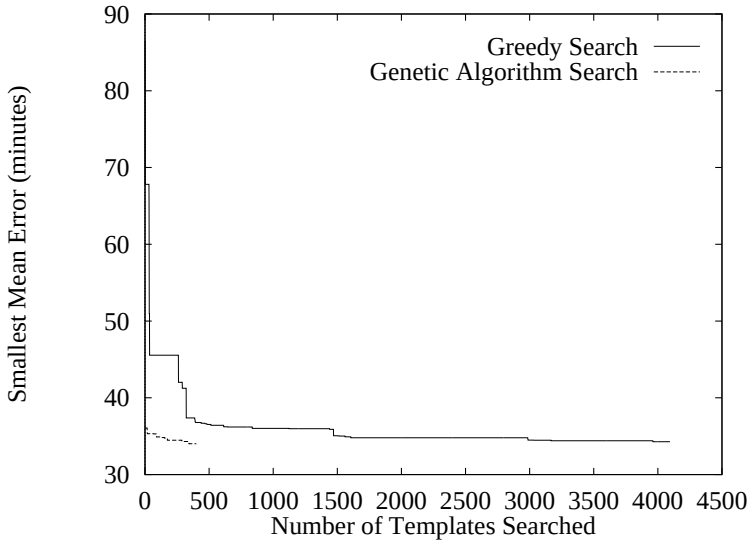

Figure 3.4: Search efficiency for workload ANL.

performed to evaluate each template set.

3.3 Related Work

Gibbons [31, 32] also uses historical information to predict the run times of parallel applications. His technique differs from ours principally in that he uses a fixed set of templates and different characteristics to define templates. Gibbons produces predictions by examining categories derived from the templates listed in Table 3.3, in the order listed, until a category that can provide a valid prediction is found. This prediction is then used as the run-time prediction.

The set of templates listed in Table 3.3 results because Gibbons uses templates of `(u,e)`, `(e)`, and `()` with subtemplates in each template. The subtemplates use the characteristics `n` and `rtime` (how long an application has executed). In this chapter, we use the user, executable, and nodes characteristics but not the running time. In this chapter, we are predicting the execution times of applications before they begin

Table 3.11: Comparison of our prediction technique with that of Gibbons.

Workload	Gibbons's Mean Error (minutes)	Our Mean Error	
		Greedy Search (minutes)	Genetic Algorithm (minutes)
ANL	75.26	34.28	34.52
CTC	124.06	117.97	98.28
SDSC95	74.05	48.33	43.20
SDSC96	122.55	50.14	47.47

to execute so this characteristic has no value. We use the running time characteristic in later chapters when run-time predictions are made after applications have started executing. Gibbons also uses the requested number of nodes slightly differently from the way we do: rather than having equal-sized ranges specified by a parameter, as we do, he defines the fixed set of exponential ranges 1, 2-3, 4-7, 8-15, and so on.

Another difference between Gibbons's technique and ours is how he performs a linear regression on the data in the categories `(u,e)`, `(e)`, and `()`. These categories are used only if one of their subcategories cannot provide a valid prediction. A weighted linear regression is performed on the mean number of nodes and the mean run time of each subcategory that contains data, with each pair weighted by the inverse of the variance of the run times in their subcategory.

Table 3.11 compares the performance of Gibbons's technique with our technique. Using code supplied by Gibbons, we applied his technique to our workloads. We see that our greedy search results in templates that perform between 4 and 59 percent better than Gibbons's technique and our genetic algorithm search finds template sets that have between 21 and 61 percent lower mean error than the template sets Gibbons selected.

In his original work, Gibbons did not have access to workloads that contained

Table 3.12: Comparison of our prediction technique to that of Gibbons, when Gibbons's technique is modified to use run times divided by maximum run times as data points

Workload	Gibbons's Mean Error (minutes)	Our Mean Error	
		Greedy Search (minutes)	Genetic Algorithm (minutes)
ANL	49.47	34.28	34.52
CTC	107.41	117.97	98.28

the maximum run time of applications, so he could not use this information to refine his technique. In order to study the potential benefit of this data on his approach, we reran his predictor while using application run time divided by the user-specified maximum run time. Table 3.12 shows our results. Using maximum run times improves the performance of Gibbons's prediction technique on both workloads, although not to the level of the predictions found during our searches.

Downey [16] uses a different technique to predict the execution time of parallel applications. His procedure is to categorize all applications in the workload, then model the cumulative distribution functions of the run times in each category, and finally use these functions to predict application run times. Downey categorizes applications using the queues that applications are submitted to, although he does state that other characteristics can be used in this categorization.

Downey observed that the cumulative distributions can be modeled by using a logarithmic function: $\beta_0 + \beta_1 \ln t$, although this function is not completely accurate for all distributions he observed. Once the distribution functions are calculated, he uses two different techniques to produce a run-time prediction. The first technique uses the median lifetime given that an application has executed for a time units. If one assumes the logarithmic model for the cumulative distribution, this equation is

$$\sqrt{ae^{\frac{1.0-\beta_0}{\beta_1}}}.$$

The second technique uses the conditional average lifetime

$$\frac{t_{max} - a}{\log t_{max} - \log a}$$

with $t_{max} = e^{(1.0-\beta_0)/\beta_1}$.

The performance of both of these techniques are shown in Table 3.13. We have reimplemented Downey's technique as described in [16] and used his technique on our workloads. The predictions are made assuming that the application being predicted has executed for one second. The data shows that of Downey's two techniques, using the median has better performance in general, and the template sets found by our genetic algorithm perform 45 to 64 percent better than Downey's best predictors. There are two reasons for this performance difference. First, our techniques use more characteristics than just the queue name to determine which applications are similar. Second, calculating a regression to the cumulative distribution functions minimizes the error for jobs of all running times while we concentrate on accurately predicting jobs with a running time of 0.

Iverson et. al. [41] and Kapadia et. al. [42] have recently published work on predicting the execution times of parallel applications. Both techniques identify application runs by a set of characteristics or features and use statistical analysis of historical data to produce predictions. Their techniques differ from ours in that they do not categorize applications explicitly. Instead, they define a distance function and use the historical applications that are close to the application being predicted using this distance function to form a prediction. They use various equations to form predictions from the k nearest neighbors, such as averaging the k neighbors,

Table 3.13: Comparison of our prediction technique with that of Downey

	Downey's Mean Error		Our Mean Error	
Workload	Conditional Median Lifetime (minutes)	Conditional Average Lifetime (minutes)	Greedy Search (minutes)	Genetic Algorithm (minutes)
ANL	97.01	106.80	34.28	34.52
CTC	179.46	201.34	117.97	98.28
SDSC95	82.44	171.00	48.33	43.20
SDSC96	102.04	168.24	50.14	47.47

performing a weighted average, or a polynomial regression. Iverson also incorporates features of the machines running the applications into the feature vectors of the applications. This allows historical run times on one machine to be used to predict the execution times of applications on another.

The run-time predictions of these two groups are very similar to ours if you express our technique as a different method for defining distance in the space of historical applications and not having a fixed value for k, but considering all applications within a certain distance. We also do not perform weighted averages or regressions when computing predictions. From the results presented by Kapadia, these weighting techniques could possibly improve the performance of our run-time prediction technique. Another difference is the amount of time it takes to make a prediction. To compute a prediction using the k nearest neighbors algorithm, it takes $O(n(p + \log n))$ time where p is number of features used to describe each application and n is the number of points in the historical database. At worst, if we use a category that contains all applications, our technique takes $O(n)$ time to form a prediction since we use a constant number of templates to form categories.

The Network Weather Service (NWS) [62] is another example of a project to use statistical techniques to forcast properties of systems. It was originally targeted to predicting network performance but it can also be used to predict other properties of systems such as application execution times. There are several differences between the prediction techniques of the NWS and our work. First, different statistical techniques are used. Second, the NWS is targeted to predicting a characteristic when given a historical time series of values for that characteristic. It does not attempt to use other characteristics that may be available for the data in the way that we do to try to find the best historical data points to use to form a prediction.

This may limit the performance of the NWS predictions in certain cases, such as predicting application execution time, when there is information available that is not used by the NWS.

3.4 Predictions in Practice

The searches we performed in previous sections evaluate a template set using all of the applications in a workload and the error is reported when predicting the same set of applications. This does not match what happens in practice. In practice, a template set is chosen using some number of applications that have already executed and the template set is then used to predict the run times of applications that are submitted in the future. We chose this approach for several reasons. First, this is the same procedure used by Downey when he calculated cumulative distribution functions and we wanted a direct comparison to his work. Second, determining the optimal amount of history to use when searching and how long to use a template set dramatically increases the amount of time it takes to find the template sets that should be used to predict application run times.

In this section, we present prediction performance results for the ANL workload when 7, 14, or 28 days of data are searched over to find template sets and then these template sets are used to predict run times for 7, 14, or 28 days. This process is repeated for every prediction interval so that all applications except those in the first 7, 14, or 28 day prediction interval are predicted. We search for templates using a genetic algorithm search and insert the data that was trained over into the historical database before predictions begin. The performance for various training set sizes and usage times are shown in Table 3.14.

Table 3.14: Performance of our run-time prediction techniques on the ANL workload with different training set sizes and lengths of use.

Training Length (days)	Time of Use (days)	Mean Prediction Error (minutes)
all	all	34.28
7	7	37.89
14	14	40.86
28	28	38.43

The first row in the table shows the performance we achieved when we searched for templates over the entire ANL workload and then used these templates to predict application run times for the same applications. Comparing this performance to the data in the rest of the table seems to indicate that we gained at least an 11 percent benefit in performance by searching over the entire workload to find our templates. However, there are two factors this data does not show. First, the most accurate predictions happen to occur at the beginning of the workload. Since there is no data before the start of the workload to search over, predictions cannot be made at the start of the workload and therefore are not incorporated into the new mean errors presented here. Second, we are only creating a historical database using the data from the interval that we trained over. If we used all of the historical data up to the end of the training set, we might achieve more accurate predictions. These two factors indicate that the advantage we gained by searching for templates over the entire workload is less than it appears in the table.

3.5 Summary

We have described a novel technique for using historical information to predict the run times of parallel applications. Our technique is to derive a prediction for a job from the run times of previous jobs judged similar by a template of key job characteristics. The novelty of our approach lies in the use of search techniques to find the best templates. We experimented with the use of both a greedy search and a genetic algorithm search for this purpose, and we found that the genetic search performs better for every workload and finds templates that result in prediction errors of 29 to 54 percent of mean run times in four supercomputer center workloads. The greedy search finds templates that result in prediction errors of 30 to 65 percent of mean run times. Furthermore, these templates provide more accurate run-time estimates than the techniques of other researchers: we achieve mean errors that are 21 to 61 percent lower error than those obtained by Gibbons and 45 to 64 percent lower error than Downey.

We find that using user guidance in the form of user-specified maximum run times when performing predictions results in a significant 19 percent to 48 percent improvement in performance for the Argonne and Cornell workloads. This suggest a simple way to greatly improve prediction performance: ask users to provide their own predictions and use this information when calculating predictions. We used both means and three types of regressions to produce run-time estimates from similar past applications and found that means are the single most accurate predictor but using a combination of predictors improves performance. For the best templates found in the greedy search, using the mean for predictions resulted in between 2 percent and 18 percent smaller errors, and using all predictors resulted in no improvement to 11 percent improvement.

Our work also provides insights into the job characteristics that are most useful for identifying similar jobs. We find that the names of the submitting user and the application are the most important characteristics to know about jobs. Predicting based on the number of nodes only improves performance by 2 to 9 percent. We also find that there is temporal locality, and hence specifying a maximum history improves prediction performance by 14 to 34 percent.

3.6 Future Work

There are several possible areas of future work. First, our prediction techniques could be used along with more detailed descriptions of applications. For example, users could identify important characteristics of their applications to the run-time predictor. These characteristics are most likely stored in configuration files or given on the command line in such a way that our current techniques cannot identify them. Identifying characteristics such as mesh size, input size, number of iterations, unique identities for input data, and so forth could result in more accurate predictions. Since these characteristics would be unique to an application, we would modify our search techniques to search over the characteristics of each application separately to find the best way to use these characteristics to form run-time predictions. The large number of search spaces, but small search spaces, could possibly reduce the time it takes to search for template sets.

A second area of future work is to extend our run-time prediction technique to metacomputing applications. If we can predict how long an application would take to complete on different sets of resources, this would greatly assist the user in selecting which sets of resources to use to execute their applications.

A third area of future work is to use our run-time prediction technique to predict other events. Our technique can be used to predict properties of events that are defined by any set of characteristics. For example, we could take data on automobile accidents, characterize the drivers of the automobiles and the automobiles themselves, and search for the best ways to group drivers and their automobiles so that we can predict how many accidents they will be in or the damages that will occur. We will describe one example of using our technique for different events, predicting queue wait times, in Chapter 4.

Chapter 4

Wait-Time Prediction

On many high-performance computers, a request to execute an application is not serviced immediately but is placed in a queue and serviced only when the necessary resources are released by running applications. On such systems, predictions of how long queued requests will wait before being serviced are useful for a variety of tasks. For example, predictions of queue wait times can guide a user in selecting an appropriate queue or, in a metacomputing environment, to an appropriate computer [30]. Wait-time predictions are also useful in a metacomputing environment when trying to submit multiple requests so that the requests all receive resources simultaneously [13]. A third use of wait-time predictions is to plan other activities in conventional supercomputing environments.

We examine two different techniques for predicting how long applications wait until they receive resources in this environment. Our first technique for predicting wait times in scheduling systems is to predict the execution time for each application in the system (using the techniques presented in Chapter 3 and [55]) and then use

those predicted execution times to drive a simulation of the scheduling algorithm. This allows us to determine the start time of every job in the scheduling system. The advantage of this technique is that, with certain scheduling algorithms and accurate run-time predictions, it can potentially provide very accurate wait-time predictions. A disadvantage is that if the scheduling algorithm is such that the start times of applications in the queues depend on applications that have not yet been submitted to the queues, the wait-time predictions will not be very accurate. A second disadvantage of this technique is that it requires detailed knowledge of the scheduling algorithm used by the scheduling system. We use four workloads recorded from supercomputers to evaluate our techniques. We find that our first technique has prediction errors of between 30 and 59 percent of mean wait times. In previous work we showed that this error is significantly better than when the run-time prediction techniques of other researchers are used [56].

Our second wait-time prediction technique predicts the wait time of an application by using the wait times of applications in the past that were in a similar scheduler state. For example, if an application is in a queue with four applications ahead of it and three behind it, how long did applications in this same state wait in the past? This approach uses the same mechanisms as our approach to predicting application execution times with different characteristics used to describe the events we are predicting. This technique has prediction errors of between 49 and 94 percent of mean wait times and is 42 percent worse than the first technique.

The remainder of this chapter is organized as follows. Section 4.1 describes the scheduling algorithms we consider in this chapter. Then, Sections 4.2 and 4.3 describes our two queue wait time prediction techniques and presents their performance.

4.1 Scheduling Algorithms

We use three basic scheduling algorithms in this work: first-come first-served (FCFS), least work first (LWF), and conservative backfill [44, 21] with FCFS queue ordering. In the FCFS algorithm, applications are given resources in the order in which they arrive. The application at the head of the queue runs whenever enough nodes become free. The LWF algorithm also tries to execute applications in order, but the applications are ordered in increasing order using estimates of the amount of work (number of nodes multiplied by estimated wallclock execution time) the application will perform.

The backfill algorithm is a variant of the FCFS algorithm. The difference is that the backfill algorithm allows an application to run before it would in FCFS order if it will not delay the execution of applications ahead of it in the queue (those that arrived before it). When the backfill algorithm tries to schedule applications, it examines every application in the queue, in order of arrival time. If an application can run (there are enough free nodes and running the application will not delay the starting times of applications ahead of it in the queue), it is started. If an application cannot run, nodes are "reserved" for it at the earliest possible time. This reservation is only to make sure that applications behind it in the queue do not delay it; the application may actually start before the reservation time.

4.2 Predicting Queue Wait Times: Technique 1

Our first wait-time prediction technique simulates the actions performed by a scheduling system using predictions of the execution times of the running and waiting applications. We simulate the FCFS, LWF, and backfill scheduling algorithms and

predict the wait time of each application when it is submitted to the scheduler.

4.2.1 Results

Table 4.1 shows the wait-time prediction performance when actual run times are used during prediction. No data is shown for the FCFS algorithm because there is no error when computing wait-time predictors in this case. The reason is that later arriving jobs do not affect the start times of the jobs that are currently in the queue. For the LWF and backfill scheduling algorithms, wait-time prediction error does occur because jobs that have not been enqueued can affect when the jobs currently in the queue can run. This effect is larger for the LWF results where later-arriving jobs that wish to perform smaller amounts of work move to the head of the queue. As one can see in the table, the wait-time prediction error for the LWF algorithm is between 34 and 43 percent: there is a very high built-in error when predicting queue wait times of the LWF algorithm with this technique. There is also a small error (3 - 4%) when predicting the wait times for the backfill scheduling algorithm.

Table 4.2 shows the wait-time prediction errors while using maximum run times as run-time predictions. The wait-time prediction error of the LWF algorithm when using actual run times as run-time predictors is 59 to 80 percent better than the wait-time prediction error when using maximum run times as the run-time predictor. For the backfill algorithm, using maximum run times results in between 96 and 99 percent worse performance than using actual run times. Maximum run times are used to predict run times in scheduling systems such as EASY [44]. These predictions are provided in the ANL and CTC workload and are implied in the SDSC workloads because each of the queues in the two SDSC workload has maximum limits on

Table 4.1: Wait-time prediction performance using actual run times.

Workload	Scheduling Algorithm	Wait-Time Prediction Mean Error (minutes)	Wait-Time Prediction Percentage of Mean Wait Time
ANL	LWF	37.14	43
ANL	Backfill	5.84	3
CTC	LWF	4.05	39
CTC	Backfill	2.62	10
SDSC95	LWF	5.83	39
SDSC95	Backfill	1.12	4
SDSC96	LWF	3.32	42
SDSC96	Backfill	0.30	3

resource usage. To derive maximum run times for the SDSC workloads, we find the longest running job in each queue and use that as the maximum run time for all jobs in that queue. The maximum run times are provided explicitly or implicitly in the workloads so they are available for use as run-time predictors and can be considered as an upper bound on run-time prediction performance.

Table 4.3 shows that our run-time prediction technique results in run-time prediction errors that are from 33 to 86 percent of mean application run times and wait-time prediction errors that are from 34 to 77 percent of mean wait times. The best wait-time prediction performance occurs for the ANL workload and the worst for the SDSC96 workload. This is the opposite of what we expect from the run-time prediction errors. The most accurate run-time predictions are for the SDSC96 workload. This implies that accurate run-time predictions are not the only factor that determines the accuracy of wait-time predictions.

The results when using our run-time predictor also show that the mean wait time prediction error is 19 to 42 percent worse than when predicting wait times for the LWF algorithm using actual run times. Finally, using our run-time predictor

Table 4.2: Wait-time prediction performance using maximum run times.

Workload	Scheduling Algorithm	Run-Time Prediction		Wait-Time Prediction	
		Mean Error (minutes)	Percentage of Mean Run Time	Mean Error (minutes)	Percentage of Mean Wait Time
ANL	FCFS	99.23	102	996.67	186
ANL	LWF	203.53	208	97.12	112
ANL	Backfill	134.82	138	429.05	242
CTC	FCFS	243.97	143	125.36	128
CTC	LWF	254.57	149	9.86	94
CTC	Backfill	264.36	154	51.16	190
SDSC95	FCFS	393.31	363	162.72	295
SDSC95	LWF	356.50	329	28.56	191
SDSC95	Backfill	377.50	349	93.81	333
SDSC96	FCFS	394.66	236	47.83	288
SDSC96	LWF	397.30	238	14.19	180
SDSC96	Backfill	397.58	238	39.66	350

results in 53 to 86 percent better wait time predictions than when using maximum run times as the run-time predictors.

4.3 Predicting Queue Wait Times: Technique 2

Our second wait-time prediction technique uses historical information on scheduler state to predict how long applications will wait until they receive resources. This is an instance of the same general prediction approach that we use to predict application run times in Chapter 3. When we predict application run times, there is only a limited set of application characteristics that are provided in the trace data. We therefore used all of the characteristics provided to describe the applications. There are no already-defined characteristics of scheduler state so we chose what we believe to be an appropriate set. These characteristics are described in Table 4.4 and describe the parallel computer being scheduled, the application whose wait time is being predicted, the time the prediction is being made, the applications that are waiting in the queue ahead of the application being predicted, and the applications that are running.

Table 4.5 shows the performance of this wait-time prediction technique. The data shows that the wait-time prediction error is 42 percent worse on average than our first wait-time prediction technique. One trend to notice is that the predictions for the FCFS scheduling algorithm are the most accurate for all of the workloads, the predictions for the backfill algorithm are all the second most accurate, and the predictions for the LWF algorithm are the least accurate. This is the same pattern when the first wait-time prediction technique is used with actual run times. This indicates that our second technique is also affected by not knowing what applications

Table 4.3: Wait-time prediction performance of our first technique.

Workload	Scheduling Algorithm	Run-Time Prediction		Wait-Time Prediction	
		Mean Error (minutes)	Percentage of Mean Run Time	Mean Error (minutes)	Percentage of Mean Wait Time
ANL	FCFS	38.26	39	161.49	30
ANL	LWF	54.11	55	44.75	51
ANL	Backfill	46.16	47	75.55	43
CTC	FCFS	125.69	73	30.84	31
CTC	LWF	145.28	85	5.74	55
CTC	Backfill	145.54	85	11.37	42
SDSC95	FCFS	53.14	49	20.34	37
SDSC95	LWF	58.98	55	8.72	58
SDSC95	Backfill	57.87	53	12.49	44
SDSC96	FCFS	55.92	33	9.74	59
SDSC96	LWF	54.27	33	4.66	59
SDSC96	Backfill	54.82	33	5.03	44

Table 4.4: Characteristics of scheduler state.

Characteristic of	Characteristic
Machine	Number of free nodes
Application	Number of nodes
Application	Maximum run time
Application	Num nodes * Max run time
Application	Waiting time
Prediction time	Time of day
Prediction time	Time of month
Waiting jobs ahead	Number of jobs
Waiting jobs ahead	Sum(nodes * max rt)
Waiting jobs ahead	Sum(nodes * pred rt)
Running jobs	Number of jobs
Running jobs	Sum(nodes * max rt)
Running jobs	Sum(nodes * pred rt)

will be submitted in the near future. Also, the wait-time prediction error using our second prediction technique for the FCFS algorithm is significantly better than the prediction error for the LWF and backfill scheduling algorithms. This seems to indicate that the characteristics we chose to represent scheduler state are better for representing the FCFS scheduler state than the state of the other two scheduling algorithms.

4.4 Summary

In this chapter, we use two techniques to predict how long applications wait before receiving resources from scheduling systems. These predictions are useful for selecting a queue or parallel computer, when seeking to obtain access to resources from multiple computers, and scheduling other activities.

Our first technique for predicting queue wait times is to run scheduling simula-

Table 4.5: Wait-time prediction performance of our second technique.

Workload	Scheduling Algorithm	Wait-Time Prediction	
		Mean Error (minutes)	Percentage of Mean Wait Time
ANL	FCFS	260.36	49
ANL	LWF	76.78	88
ANL	Backfill	130.35	74
CTC	FCFS	76.18	78
CTC	LWF	9.80	94
CTC	Backfill	22.95	85
SDSC95	FCFS	39.79	72
SDSC95	LWF	13.67	91
SDSC95	Backfill	25.50	90
SDSC96	FCFS	10.55	64
SDSC96	LWF	6.83	87
SDSC96	Backfill	9.26	82

tions using predictions of application execution times. Our technique for predicting application run times is to derive a prediction for an application from the run times of previous applications judged similar by a template of key job characteristics. The novelty of our approach lies in the use of search techniques to find the best templates. The advantage of this approach is that it results in more accurate wait-time predictions than the second approach. The disadvantages are that it requires knowledge of the scheduling algorithm and has an inherent limitation because it does not consider the effect of applications that have not been enqueued yet. The wait-time prediction errors of our first wait-time prediction technique are from 30 to 59 percent of mean wait times. This is 74 percent better on average than when using maximum run times as the run-time predictors, moreover, our previous work has shown that it is significantly better than when using the run-time predictors of other researchers.

Our second wait-time prediction technique is to use the wait times experienced by applications in similar past scheduling states to predict how long an application will wait until it receives resources. We find that this technique has the advantage of operating without knowledge of the scheduling algorithm being used but it has lower performance than our first technique. We find that this technique has wait-time prediction errors that are 49 to 94 percent of mean wait times and are on average 42 percent worse than the prediction errors of our first technique.

4.5 Future Work

There are several possible ways to improve the performance of our wait-time prediction techniques in the future. The performance of our first technique that predicts application run times and performs scheduler simulations could be improved by analyzing the arriving applications. Our scheduler simulator could use a synthetic workload of applications that will arrive in the future to approximate the applications that will arrive and improve it's wait time prediction performance when applications that have not arrived yet can affect scheduling decisions. Our second wait time prediction technique could possibly be improved by studying what characteristics should be used to describe scheduler state when predicting wait times.

Another possible way to improve wait time prediction performance is to combine our two techniques. If our first prediction technique is producing errors in predictable ways (one case where this might occur is when applications that have not been submitted can affect the scheduling of applications that have already been submitted), then we could attempt to adjust the predictions to reduce the error. One ideal way to do this is to have the prediction made by the first technique be

one of the characteristics used by our second wait time prediction technique. The prediction made by the second technique would then be given to the user.

Chapter 5

Scheduling with Predictions

Many scheduling algorithms use predictions of application execution times when making scheduling decisions [44, 21, 19]. In this chapter, we investigate if our run-time predictions result in better schedules when used with two commonly used scheduling techniques. These algorithms use run-time predictions when making scheduling decisions, and we therefore expect that more accurate run-time predictions will improve scheduling performance. Using our run-time predictions and our four workloads recorded from the Argonne SP, Cornell SP, and the SDSC Paragon, we find that the accuracy of the run-time predictions has a minimal effect on the utilization of the systems we are simulating. We also find that using our run-time predictors results in mean wait times that are within 22 percent of the mean wait times that are obtained if the scheduler exactly knows the run times of all of the applications. When comparing the different predictors, our run-time predictor results in 2 to 67 percent smaller mean wait times for the workload with the highest offered load. No prediction technique clearly outperforms the other techniques when the

offered load is low.

The next section will describe how we find the best templates to use to predict application run times when scheduling. Section 5.2 presents the performance of our run-time predictions and compares this performance to the scheduling performance when other run-time predictions are used.

5.1 Run-Time Prediction Experiments

The first thing we need to determine is what template sets to use to predict application run times. To begin with, we will assume that the best way to optimize scheduling performance is to minimize run-time prediction error. Therefore, we need to determine what run-time predictions and insertions of run times to search over to find the best template sets to use. This depends on the workloads and scheduling algorithms we use. We use our four workloads recorded from the ANL SP, CTC SP, and SDSC Paragon and the LWF and backfill scheduling algorithms described in Section 4.1.

When using run-time predictions while scheduling, run-time predictions are also made at different times for each algorithm/trace pair, and we attempt to find the optimal template sets to use for each pair. The FCFS algorithm does not use run-time predictions when scheduling, so we only consider the LWF and backfill algorithms here. For the LWF algorithm, all waiting applications are predicted whenever the scheduling algorithm attempts to start an application (when any application is enqueued or finishes). This occurs because the LWF algorithm needs to find the waiting application that will use the least work. For the backfill algorithm, all running and waiting applications are predicted whenever the scheduling algorithm attempts

to start an application (when any application is enqueued or finishes).

We generate our run-time prediction workloads for scheduling using maximum run times as run-time predictions. We note that using maximum run times will produce predictions and insertions slightly different from those produced when the LWF and backfill algorithms use other run-time predictions. Nevertheless, we believe that these run-time prediction workloads are representative of the predictions and insertions that will be made when scheduling using other run-time predictors.

We also use a second technique to find the best templates to use to predict run times. Instead of attempting to minimize run-time prediction error so that scheduling performance is maximized, we perform scheduling simulations and attempt to directly minimize wait times. We do not attempt to maximize utilization because utilization only changes very slightly when different template sets are used or even when a different scheduling algorithm is used.

5.2 Results

Our goal in this chapter is to improve the performance of the LWF and backfill scheduling algorithms. Table 5.1 shows the performance of the scheduling algorithms when the actual run times are used as run-time predictors. This is the best performance we can expect in each case and serves as an upper bound on scheduling performance.

Table 5.2 shows the performance of using maximum run times as run time predictions in terms of average utilization and mean wait time. The scheduling performance when using the maximum run times can once again be considered an upper bound for comparison. When comparing this data to the data in Table 5.1, one

Table 5.1: Scheduling performance using actual run times.

Workload	Scheduling Algorithm	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	70.34	61.20
ANL	Backfill	71.04	142.45
CTC	LWF	55.18	11.15
CTC	Backfill	55.18	27.11
SDSC95	LWF	41.14	14.48
SDSC95	Backfill	41.14	21.98
SDSC96	LWF	46.79	6.80
SDSC96	Backfill	46.79	10.42

can see that the maximum run times are an inaccurate predictor but this fact does not affect the utilization of the simulated parallel computers. Predicting run times with actual run-times when scheduling results in 3 to 27 percent lower mean wait times, except in one case where using maximum run times results in 6 percent lower mean wait times. The effect of accurate run-time predictions is highest for the ANL workload which has the largest offered load.

Table 5.3 shows the performance of using our run-time prediction technique when scheduling with the template sets found by the searches that were minimizing run-time prediction error. The run-time prediction error in this case is between 22 and 119 percent of mean run times, slightly worse than the results when predicting run-times for wait-time prediction. This worse performance is due to more predictions being performed. First, more predictions are made of applications before they begin executing; and these predictions do not have information about how long an application has executed. Second, more predictions are made of long-running applications, the applications that contribute the largest errors to the mean errors.

When scheduling using our run-time prediction technique, the mean wait times

Table 5.2: Scheduling performance using maximum run times.

Workload	Scheduling Algorithm	Run-Time Prediction		Scheduling Performance	
		Mean Error (minutes)	Percentage of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	104.12	107	70.70	83.81
ANL	Backfill	154.86	158	71.04	177.14
CTC	LWF	378.91	207.47	70.63	36.95
CTC	Backfill	183.76	100.62	70.63	123.91
SDSC95	LWF	411.47	380	41.14	14.95
SDSC95	Backfill	377.50	349	41.14	28.20
SDSC96	LWF	397.30	238	46.79	7.88
SDSC96	Backfill	387.64	232	46.79	11.34

Table 5.3: Scheduling performance using our run-time prediction technique (first template set).

		Run-Time Prediction		Scheduling Performance	
Workload	Scheduling Algorithm	Mean Error (minutes)	Percent of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	60.44	62	70.22	84.30
ANL	Backfill	116.38	119	71.04	177.23
CTC	LWF	176.06	96	55.18	13.37
CTC	Backfill	56.96	31	55.18	29.73
SDSC95	LWF	62.70	58	41.14	16.67
SDSC95	Backfill	56.79	52	41.14	51.25
SDSC96	LWF	73.75	44	46.79	7.87
SDSC96	Backfill	38.16	22	46.79	10.67

that are always worse than when using actual run times as predictions, and are 19 percent worse on average. There is very little difference in the utilizations when our run-time predictions are used instead of actual run times. The mean wait times achieved by the scheduling algorithms when using our run-time predictions are 7 percent smaller on average than when predicting using maximum run times. Our run-time predictions increase performance the most for the CTC workload and decrease performance the most for the SDSC95 workload.

We were not very satisfied with the scheduling performance achieved when using our previous run-time predictions. To attempt to improve scheduling performance, we perform new searches. Instead of trying to minimize run-time prediction error, we attempt to minimize mean wait times. Table 5.4 shows the performance of the results of these searches. If this data is compared to the data in Table 5.3, one can see that the wait time is improved in all cases by an average of 14 percent. Further, the run time prediction error is larger for all but one of the template sets found during the second search. For the backfill scheduling algorithm, the second search finds run-time prediction templates that are 2 percent worse on average (one of the cases is 48 percent better). For the LWF scheduling algorithm, the second search finds run-time prediction templates that have 27 percent higher run-time prediction errors. This shows that for the backfill algorithm there is some correlation between run-time prediction accuracy and scheduling performance but this is not the case for the LWF scheduling algorithm.

When comparing our run-time prediction technique to using maximum run times, our technique has a minimal effect on the utilization of the systems, but it does decrease the mean wait time in six of the eight experiments. Table 5.5 through Table 5.7 show the performance of the scheduling algorithms when using Gibbons's

Table 5.4: Scheduling performance using our run-time prediction technique (second template set).

		Run-Time Prediction		Scheduling	
Workload	Scheduling Algorithm	Mean Error (minutes)	Percent of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	72.88	74	70.75	69.31
ANL	Backfill	116.52	170	71.04	174.14
CTC	LWF	182.96	100	55.18	10.58
CTC	Backfill	61.73	33	55.18	28.39
SDSC95	LWF	142.01	131	41.14	14.82
SDSC95	Backfill	38.37	35	41.14	22.21
SDSC96	LWF	112.13	67	46.79	7.43
SDSC96	Backfill	47.06	28	46.79	10.56

and Downey's run-time predictors. The results indicate that once again, using our run-time predictor does not produce greater utilization. The results also show that our run-time predictor results in 1 to 23 percent lower mean wait times than Gibbons' predictor. There is no trend to indicate that our run-time predictor is better for a particular workload or scheduling algorithm. If we compare the scheduling performance of our run-time predictor to scheduling performance when using Downey's run-time predictor, our predictor results in much smaller wait times for the ANL workload but larger wait times when performing backfill scheduling of the other workloads. In fact, Downey's predictors result in lower wait times when backfilling the CTC and SDSC workload than when using actual run times. One explanation for this is that over-estimating execution times gives the scheduler more room to start smaller jobs earlier, and therefore lowering the mean wait times [64].

5.3 Summary

In this chapter, we use our predictions of application run times to improve the performance of the least-work-first and backfill scheduling algorithms. We find that the utilization of the parallel computers we simulate does not vary greatly when using different run-time predictors, but using our run-time predictions does improve the mean wait times in general. In particular, our more accurate run-time predictors have the largest effect on mean wait time for the CTC workload. We also find that on average, the mean wait time when using our predictor is only 5 percent larger than the mean wait time that would occur if the scheduler knows the exact run times of the applications. The difference in wait time is largest for the ANL workload, at 15 percent, which has the highest offered load. Further, the wait times are smaller

Table 5.5: Scheduling performance using Gibbons' prediction technique.

		Run-Time Prediction		Scheduling	
Workload	Scheduling Algorithm	Mean Error (minutes)	Percent of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	92.81	95	70.72	90.36
ANL	Backfill	86.81	89	71.04	181.38
CTC	LWF	175.33	96	55.18	11.62
CTC	Backfill	101.13	55	55.18	29.85
SDSC95	LWF	132.45	122	41.14	15.99
SDSC95	Backfill	91.35	84	41.14	24.83
SDSC96	LWF	178.12	107	46.79	7.51
SDSC96	Backfill	47.56	28	46.79	10.82

Table 5.6: Scheduling performance using Downey's conditional average run-time predictor.

		Run-Time Prediction		Scheduling	
Workload	Scheduling Algorithm	Mean Error (minutes)	Percent of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	74.54	76	71.04	154.76
ANL	Backfill	144.80	148	70.88	246.40
CTC	LWF	173.81	95	55.18	10.17
CTC	Backfill	40.72	22	55.18	19.18
SDSC95	LWF	279.04	258	41.14	16.22
SDSC95	Backfill	648.84	709	41.14	20.37
SDSC96	LWF	278.46	167	46.79	7.88
SDSC96	Backfill	470.91	282	46.79	8.25

Table 5.7: Scheduling performance using Downey's conditional median run-time predictor.

		Run-Time Prediction		Scheduling	
Workload	Scheduling Algorithm	Mean Error (minutes)	Percent of Mean Run Time	Utilization (percent)	Mean Wait Time (minutes)
ANL	LWF	62.22	64	71.04	154.76
ANL	Backfill	144.11	147	71.04	207.17
CTC	LWF	158.02	87	55.18	10.90
CTC	Backfill	29.05	16	55.18	16.82
SDSC95	LWF	127.97	118	41.14	16.36
SDSC95	Backfill	347.42	321	41.14	19.56
SDSC96	LWF	138.44	83	46.79	7.80
SDSC96	Backfill	275.74	165	46.79	8.02

when using our run-time predictions instead of Gibbons'. An interesting occurance that we examine is that Downey's run-time predictor results in lower wait-times for the backfill scheduling algorithm and the CTC and SDSC workloads than our run-time predictor or even when using actual run times as run-time predictions.

5.4 Future Work

One area of future work is if schedulers have accurate run-time predictions, what should they do with them? Work presented in this chapter and in [64] indicate that if current scheduling algorithms use more accurate run-time predictions, they may not improve their scheduling performance. We could investigate new scheduling algorithms that use more accurate run-time predictions in different ways. For example, a scheduler could use maximum run times to calculate the latest time the application should start. The scheduler could then perform a schedule optimization technique such as bin packing [19] to produce an schedule that satisfies these start time guarantees but optimizes wait time, for example.

Chapter 6

Reservations

Software support for metacomputing allows users to execute applications on resources at more than one site. In many cases, an application will want simultaneous access to resources controlled by more than one entity [30]. The problem with this is that current supercomputer schedulers do not provide mechanisms to allow such simultaneous access. At the present time, a user has to either communicate with the administrators of the computers and arrange for resources to be reserved, or submit applications to queues on each computer system with no guarantee that the subapplications will execute simultaneously.

In this chapter, we investigate one solution to this *coallocation* problem: advanced reservation of resources. If a user can reserve resources ahead of time, reservations can be made on several systems at the same time. We investigate several different ways to add support for reservations into supercomputer scheduling systems and evaluate the performance. We evaluate scheduling performance using the following metrics:

- *Utilization.* The average percent of the machine that is being used by applications.

- *Mean wait time.* The average amount of time that applications wait before receiving resources.

- *Mean offset from requested reservation time.* The average difference between when the users initially want to reserve resources for each application and when they actually obtain reservations.

We used utilization and mean wait time to evaluate scheduling systems in the previous chapter. These two metrics allow us to examine the effect that support for reservations has on traditional scheduling performance. The mean offset from reservation time is a new metric and measures how well the scheduler is at satisfying reservation requests.

In this chapter, we use these metrics to evaluate a variety of techniques for combining queuing scheduling with reservation. There are several assumptions and choices to be made when doing this. The first is whether applications are restartable. Most scheduling systems currently assume that applications are not restartable (a notable exception is the Condor system [46]). Restartable applications can be supported either at the user level or the system level. If restarting is supported at the user level, the application must be written so that if it is terminated, it can be restarted without the results of the later execution(s) being affected by the partial execution. In fact, if an application saves intermediate results, it can continue execution using intermediate results. Restartable applications can be supported at the system level by providing support for checkpointing and restarting applications from checkpoints. One system that provides this support is Condor, although there

are some limits on what services applications can use.

We evaluate scheduling techniques when applications both can and cannot be restarted. We assume that when an application is terminated, intermediate results are not saved and applications must restart execution from the beginning. This assumption was made because system support for restartable applications is not widely available and we do not have models for when the applications in our workloads save intermediate results.

Another assumption we have to make is the model for reservation of resources. We assume that a running application that was reserved cannot be terminated to start another application. Further, we assume that once the scheduler agrees to a reservation time, the application will start at that time. Further details of our model are presented in Section 6.1

If we assume that applications are not restartable, then we must make sure that nodes are available for applications with reservations because we assume that once a reservation is made, the scheduler must fulfill it. To ensure that nodes are available, we must use maximum run times when predicting application execution times and the resulting scheduling algorithms essentially perform backfilling. Details of these algorithms and their performance are presented in Section 6.2.

If applications are restartable, there are more options for the scheduling algorithm and this allows us to improve the performance of scheduling systems. First, the scheduler can use run-time predictions other than maximum run times. Second, the scheduler can consider reservations when scheduling queued applications or not consider them. Third, there are many different ways to select which running applications from the queue to terminate to start a reserved application. Details of these options and their performance are presented in Section 6.3

6.1 Reservation Model

Before we describe our scheduling algorithms that support reservations, we must describe the model we use for reservations. First, in our model, a reservation request goes to a single supercomputer and consists of the number of nodes desired, the maximum amount of time the nodes will be used, the desired start time, and the application to run on those resources. Second, we assume that the following procedure occurs when a user wishes to submit a reservation request:

1. The user asks if they can run an application at time T_r on N nodes for at most M amount of time.

2. The scheduler makes the reservation at time T_r if it can. In this case, the reservation time, T, equals the requested reservation time, T_r.

3. If the scheduler cannot make the reservation at time T_r, it replies with a list of times it could make the reservation and the user picks the available reservation time T which is closest in time to T_r.

We use this procedure when performing the scheduling simulations described in the succeeding sections. Note that this procedure is realistic for what a user would do on a single system but will not exactly match what a user will do when trying to reserve resources from more than one scheduler simultaneously. When reserving on more than one system, if the user cannot reserve resources at their originally requested time T_r^{meta} on all systems, the user will examine the lists of times returned from the scheduler on each machine and pick a time T^{meta} that is available on all systems and is as close to T_r^{meta} as possible. There should be a correlation between the difference $|T - T_r|$ and $|T^{meta} - T_r^{meta}|$. That is, if we

develop a scheduling algorithm that reduces $|T - T_r|$, then $|T^{meta} - T_r^{meta}|$ should be reduced as well when the schedulers use that scheduling algorithm.

The last part of the model is what occurs when an application is terminated. First, only applications that came from a queue can be terminated. Second, when an application is terminated, it is placed back in the queue from which it came. For FCFS, the correct position in the queue for the terminated application is determined by submission time. For LWF, the correct position is determined by the predicted amount of work to be performed. Third, as was mentioned previously, when an application is terminated, we assume that all of the work it performed is lost.

6.2 Nonrestartable Applications

In this section, we assume that applications cannot be terminated and restarted at a later time and that once a reservation is agreed to by the scheduler, it must be fulfilled. A scheduler with these assumptions must not start an application from a queue unless it is sure that starting that application will not cause a reservation to be unfulfilled. Further, the scheduler must make sure that reserved applications do not execute longer than expected and prevent other reserved applications from starting.

There are two mechanisms to be described to support these constraints. The first is how the scheduler decides when an application from a queue can be started. The technique used for this is very similar to the backfill algorithm: The scheduler creates a timeline of when it believes the nodes of the system will be used in the future. First, the scheduler adds the currently running applications and the reserved applications to the timeline using their maximum run times. Then, the scheduler

attempts to start applications from the queue using the timeline and the number of nodes and maximum run time requested by the application to make sure that there are no conflicts for node use.

In this chapter, we use both FCFS and LWF queue ordering. If backfilling is not being performed, the timeline is still used when starting an application from the head of the queue to make sure that the application does not use any nodes that will be needed by reservations. Backfilling can be performed for both FCFS and LWF queue ordering. In this case, the timeline is used to try to start applications from the queue and to "reserve" nodes for the applications in the future if they cannot start. These "reservations" are not true reservations, just placeholders so that applications later in the queue will not start and delay the application. For each application that cannot start at the current time, a "reservation" is made for it at the earliest time that it can execute.

The second mechanism is how a scheduler makes a reservation. To make a reservation, the scheduler first performs a scheduling simulation of applications currently in the system and produces a timeline of when nodes will be used in the future. This timeline is then used to determine when a reservation for an application can be made. If the requested time for the reservation is not available, the scheduler presents the user with a list of times when their reservation could be made and we assume that the user picks the available time closest to their requested time. The scheduler uses maximum run times when creating the timeline. This guarantees that reserved applications do not conflict with running applications or other reserved applications.

One parameter that is used when reserving resources is the relative priorities of queued and reserved applications. For example, if queued applications have higher priority, then an incoming reservation cannot delay any of the applications in the

queues from starting. If reserved applications have higher priority, then an incoming reservation can delay any of the applications in the queue. The parameter we use is the percentage of queued applications that can be delayed by a reservation request.

A second parameter is when reservations can be requested. Reservation requests could be forced to ask for resources at least h hours in advance. We investigate the impact of this parameter by examining the scheduling performance when reservations are made from zero to two, one to three, or two to four hours in the future.

To evaluate our scheduling algorithms that support reservations, we derive six new workloads from each of the four workloads we have been using to this point. We randomly change either ten percent or twenty percent of the applications in an original workload to be reservations. We choose these percentages because we believe that the majority of applications will not need reservations to execute and policy decisions will be made so that there will be no start time advantage to making a reservation over submitting to a queue. For each reservation, we randomly set the ideal reservation time to be within either zero to two hours in the future, one to three hours in the future, or two to four hours in the future.

In the next subsections, we first examine the effects of limiting reservation requests to be at least zero, one, or two hours in the future. Second, we examine the effect reservations have on utilization and the mean wait time of applications from the queue. Third, we examine the changes in the difference between the requested reservation time and the time the reservation is actually made when different scheduling strategies are used. Fourth, we look at the changes in performance when reservations can delay more or fewer applications in the queue.

6.2.1 Effects of Reservation Time

In this section, we examine the effect on queued applications of not allowing reservations to be requested until later than some minimum time in the future. We examine this effect by randomly selecting the requested reservation times for applications in our workloads in various intervals in the future. We used three different intervals: the current time to two hours in the future, one to three hours in the future, and two to four hours in the future. We assume that a reservation cannot be made at a time that would delay the start of any application currently in the queue (queued applications have priority).

A sample of our data for the ANL workload is shown in Figure 6.1. This graph is representative of our data and shows the mean wait time for the ANL workload with various percentages of reservations and various intervals for requested reservation times, FCFS queue ordering, queued applications given priority when performing reservations, and backfilling is both used and not used. This data shows that how far in the future the requested reservation times are has little impact on the mean wait times. In fact, there is no correlation between reservations farther in the future and lower wait times for queued applications. We also find that there is no change in utilization when different intervals are used for requested reservation times.

Another sample of our data is shown in Figure 6.2. This graph is again representative of our other data and shows the mean difference between the requested reservation times specified by the applications and the actual time they reserved resources for. Once again, the data is gathered from the ANL workload with two different percentages of reservations and three different intervals for requested reservation times, FCFS queue ordering, queued applications given priority when reserving, and backfilling is both used and not used. The graphs show that the interval for

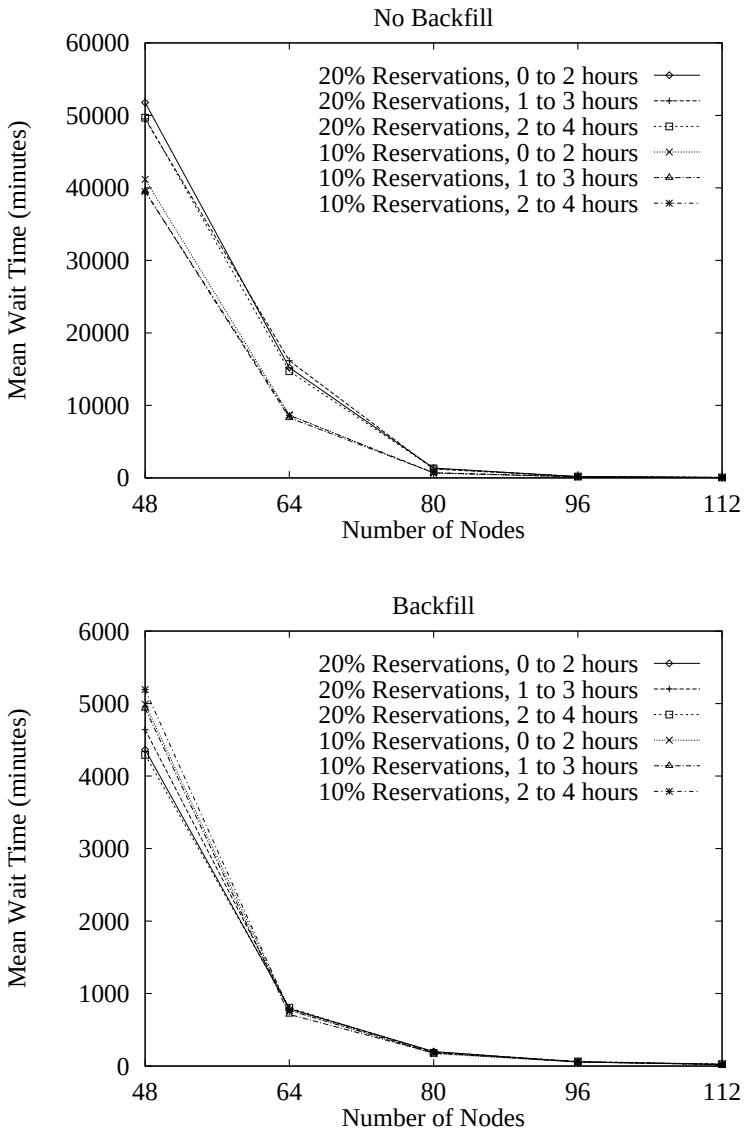

Figure 6.1: The mean wait times of queued applications for the ANL workloads with various percentages of reservations and various intervals for requested reservations, no restarting of applications, queued applications have priority, and using maximum run times to predict.

requested reservation time does affect how far reservations are from their requested reservation times, but there is no consistent correlation between interval and offset from requested reservation time. Further, most of the changes in offset are relatively small.

From the data presented here and the data we collected from our other workloads and scheduling algorithms, we determine that there is a relatively small and unpredictable effect when reservations are made in different intervals in the future. This indicates that there would be a relatively small, but unpredictable effect if a scheduler forces reservations to be made some minimum time in the future. Therefore, the use of such a minimum time should be decided by policy, not any performance effects.

For the rest of this chapter, we will only present data for workloads that have reservations with requested times zero to two hours in the future. We present this data because we believe that schedulers will not force users to reserve time later than some minimum time.

An obvious observation from Figures 6.1 and 6.2 is that the number of reservations does affect scheduling performance. The next section will discuss these effects.

6.2.2 Effect of Reservations on Scheduling

This section evaluates the impact on the mean wait times of queued applications when reservations are added to our workloads. We assume the best case for queued applications: When reservations arrive, they cannot be scheduled so that they delay any currently queued applications. First, we examine the wait times of queued applications when backfilling is not allowed.

A sample of our data is shown in Figures 6.3 and 6.4. As one can see from

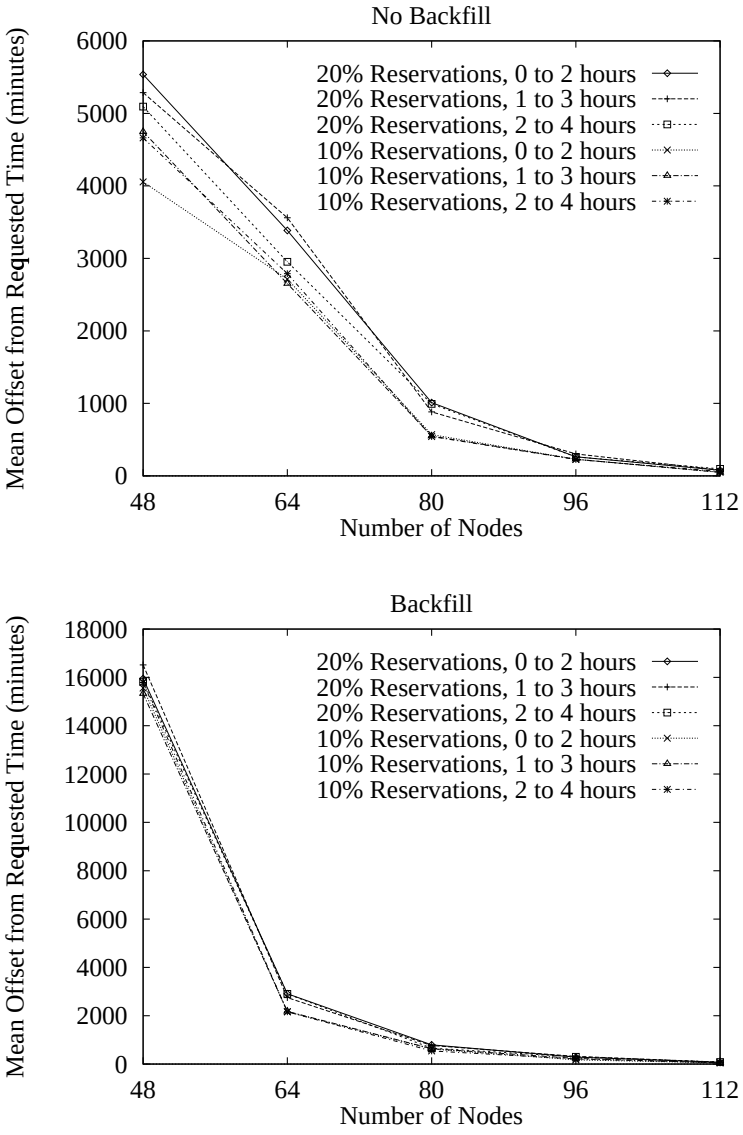

Figure 6.2: The mean difference from requested reservation times of reserved applications for the ANL workloads with various percentages of reservations and various intervals for requested reservations, no restarting of applications, queued applications have priority, and using maximum run times to predict.

the graphs, adding reservations increases the wait times of queued applications for almost all scheduling algorithms, and numbers of nodes. For the data shown in the figures, adding reservations delays queued applications from starting an average of 10 percent when ten percent of the applications are reservations and 23 percent when twenty percent of the applications are reservations. For all of the workloads, queue wait times are increased an average of 24 percent when ten percent of the applications are reservations and 144 percent when twenty percent of the applications are reservations. The large overhead when there are 20 percent reservations is due to the SDSC workloads that have very high overheads. This is most likely caused by the inaccuracy of the maximum wait times used when scheduling the SDSC workloads. Recall that the original SDSC workloads do not contain maximum run times so we computed them by finding the longest running application in each queue and using that run time as the maximum run time for all applications in that queue.

Figure 6.3 and 6.4 also show that increasing the percent of applications that are reservations in the CTC workload has roughly the same impact whether FCFS or LWF queue ordering is used. This is also true for the SDSC95 workload, but there is a larger increase in wait time with LWF queue ordering for the other two workloads. There is a small increase in wait time for the SDSC96 workload, but the increase in wait time doubles for LWF queue ordering for the ANL workload when compared to FCFS ordering.

These figures can also be used to analyze whether reservations have a larger impact on wait time when backfilling is used or not. The figures show that the wait time increases twice as much when backfilling is not performed. This also occurs for the other workloads, with the ANL increase in wait time doubling when not backfilling, and slightly smaller increases for the SDSC workloads.

Next, we use our data to analyze what happens to the mean wait times when the load on the machines increase (fewer nodes are used). We find that in all but one case, the wait times increase as the number of nodes decrease. There is a relatively linear increase in wait time for the CTC and SDSC95 workloads as the number of nodes decreases but superlinear increases for the ANL and SDSC96 workloads.

One interesting effect is shown in Figure 6.5. The first graph in the figure shows the mean wait time for the ANL workload when FCFS queue ordering is used with backfilling and maximum run times. As one can see at 48 nodes, the longest mean wait time occurs when there are no reservations and the shortest mean wait time is when there are twenty percent reservations. This can be explained by examining the large spike in the second graph in the figure at 48 nodes which shows that reservations are not being satisfied until far after their requested times. When the offered load to the scheduler is very high, reservation request are being satisfied far in the future, leaving more room for backfilling applications from the queue; and if there are fewer applications in the queue, they will have shorter waiting times.

6.2.3 Offset from Requested Reservations

In this section, we examine the difference between the requested reservation times of the applications in our workload and the times they receive their reservations. We again assume that reservations cannot be made at a time that would delay the startup of any applications in the queue at the time the reservation is made. The performance is what is expected in general: the offset is larger when fewer nodes are used and when there are more reservations. There are some cases where the offset increases superlinearly as we saw in Figure 6.5. This occurs with the ANL workload for all numbers of reservations when backfilling is performed and when 20% of the

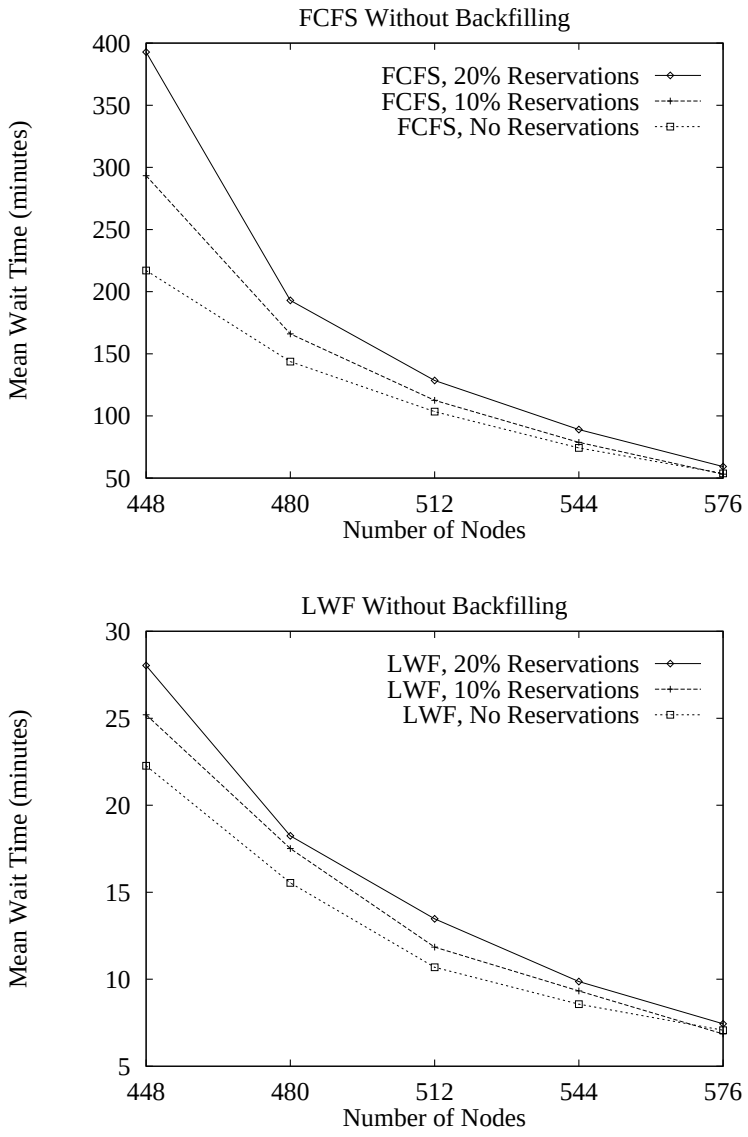

Figure 6.3: The mean wait times of queued applications for the CTC workloads with no backfilling, no restarting of applications, queued applications have priority, and using maximum run times to predict.

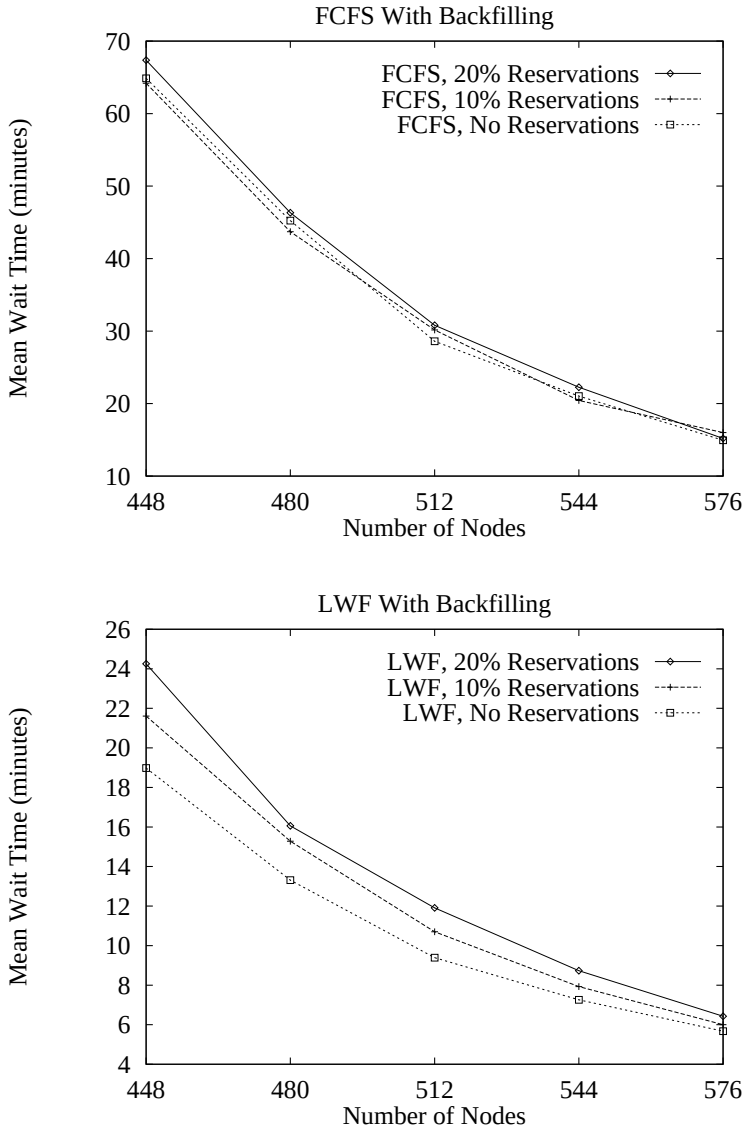

Figure 6.4: The mean wait times of queued applications for the CTC workloads with backfilling, no restarting of applications, queued applications have priority, and using maximum run times to predict.

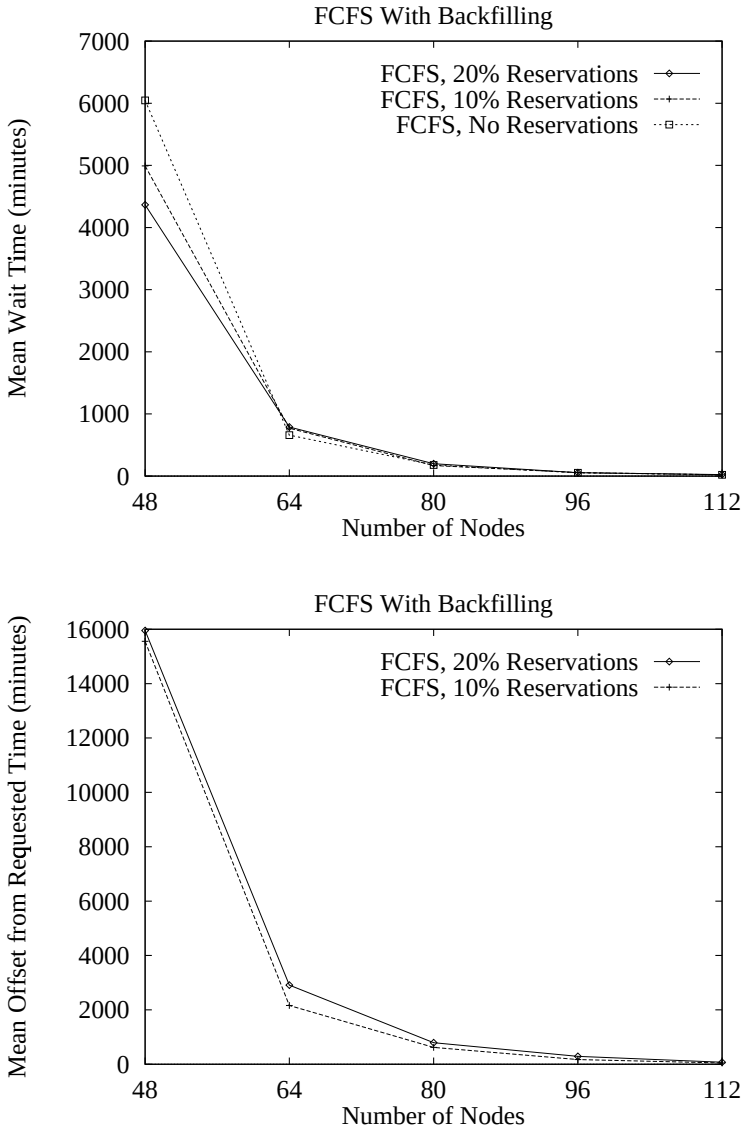

Figure 6.5: Scheduling performance of queued applications for the ANL workloads with FCFS queue ordering, backfilling, no restarting of applications, queued applications have priority, and using maximum run times to predict.

applications in the SDSC96 workload are changed to reservations. For the other cases, there is a relatively steady increase in the difference from requested reservation time as the number of nodes decreases as shown in Figure 6.6 and Figure 6.7.

The figures also show that the difference between requested reservation times and actual reservation times is larger when FCFS queue ordering is used. This holds true for the other workloads as well. A final observation is that our data shows that when backfilling is performed, the difference between requested reservation time and actual reservation time is larger, for the SDSC workloads, is the same for the CTC workload, and is smaller for the ANL workload. One can see that this pattern follows the utilization or offered load on the machines: the SDSC workloads have the lowest utilization, and the ANL workload the highest. This indicates that backfilling results in reservations closer to their requested reservations only when there are sufficient applications available for backfilling.

6.2.4 Effect of Application Priority

In this section, we examine the effects on mean wait time and the mean difference between reservation time and requested reservation time when queued applications are not given priority over all reserved applications. We accomplish this by giving zero, fifty, or one-hundred percent of queued applications priority over a reserved application when a reservation request is being made (not delaying zero, fifty, or one-hundred percent of queued applications when a reservation is made). This data for FCFS queue ordering and backfilling for the ANL workload is shown in Figure 6.8.

The data in the figure shows that there is a significant impact on both wait time and offset from requested reservation time when the number of queued applications that can be delayed by reservations is varied. As expected, if more queued

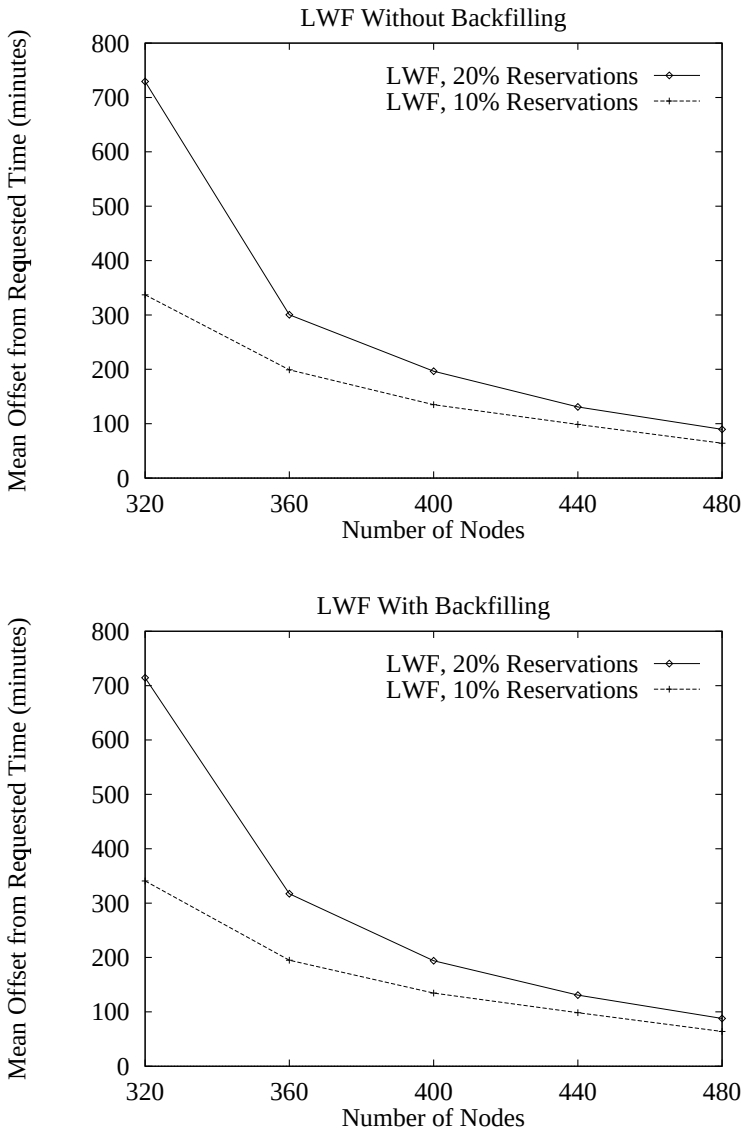

Figure 6.6: Scheduling performance of queued applications for the SDSC95 workload with LWF queue ordering, no restarting of applications, queued applications have priority, and using maximum run times to predict.

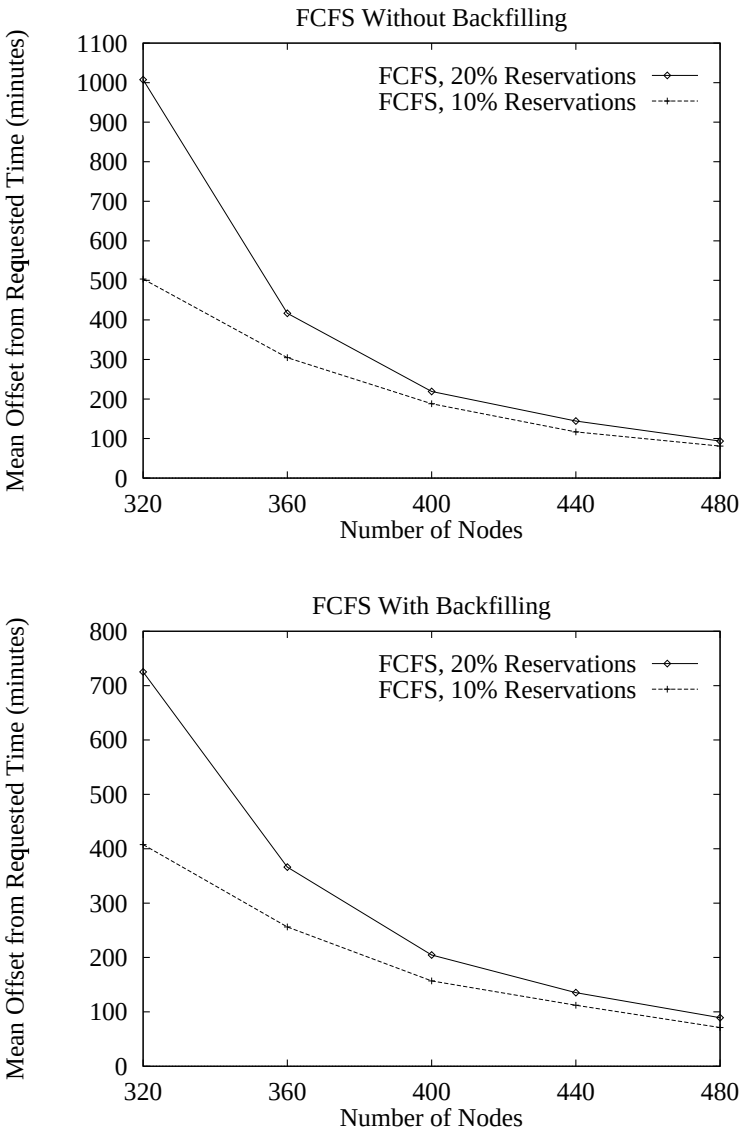

Figure 6.7: Scheduling performance of queued applications for the SDSC95 workload with FCFS queue ordering, no restarting of applications, queued applications have priority, and using maximum run times to predict.

applications can be delayed when a reservation request arrives, then the wait times are generally longer and the offsets are smaller. On average for the data shown in the figure, decreasing the percent of queued applications with priority from 100 to 50 percent increases mean wait time by 8 percent and decreases mean offset from requested reservation time by 44 percent. Decreasing the percent of queued application with priority from 100 to 0 percent increases mean wait time by 36 percent and decreases mean offset by 97 percent. These results for the change in the difference between reservation time and requested reservation time are representative of our data: as fewer queued applications have priority, the reservations are closer to their requested reservations. There is also a trend that when more queued applications have priority, the mean wait time decreases. This does not occur in all cases, and when it does not occur, the increase in wait time is small. In particular, these small increases in wait time when more queued applications have priority occur most frequently for the ANL workload, the workload with the highest offered load.

6.3 Restartable Applications

This section describes and evaluates our techniques for performing reservations assuming that running applications can be terminated and restarted at a later time. If we make this assumption, we can use run-time predictions other than maximum run times and this may allow us to improve scheduling performance. We use our run-time prediction technique with the template sets derived in Chapter 5 and evaluate several options for the scheduling algorithm. First, we evaluate different ways to select which applications to terminate if the scheduler needs nodes to satisfy a reservation. Second, we determine if reservations that must be satisfied in the near

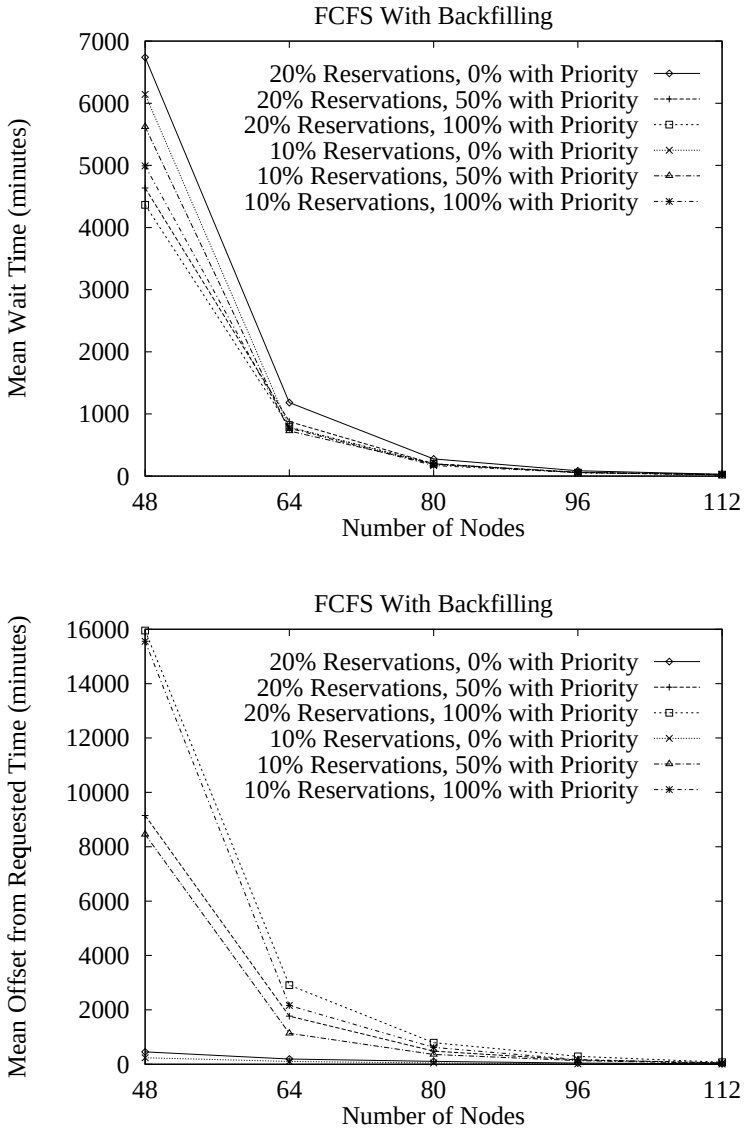

Figure 6.8: Scheduling performance for the ANL workload with FCFS queue ordering, backfilling, no restarting of applications, and using maximum run times to predict.

future should be considered when starting queued applications. Third, we compare the scheduling performance when applications can be restarted to when they cannot. In this section, we assume that reservations cannot be made at times that will delay the startup of applications in the queues.

6.3.1 Selecting Applications for Termination

There are many possible ways to select which running applications that came from a queue should be terminated to allow a reservation to be satisfied. We choose a rather simple technique where the scheduler orders running applications from queues in a list based on some cost. The applications are then terminated in increasing order of cost until enough nodes are available for the reservation to be satisfied.

We use the equation $aNT_p + bNT_f + cR$ to determine the cost of terminating each application. In the equation, a, b, and c are constants, N is the number of nodes being used by the application, T_p is the amount of time the application has executed, T_f is the amount of time the scheduler expects the application will continue to execute, and R is the number of times the application has been restarted. The motivation behind this equation is that increasing the constant a will increase the cost of terminating an application that has performed a large amount of work that would be lost, decreasing b below zero will decrease the cost of terminating the application if it still has a large amount of work to do, and increasing c will decrease the number of times any particular application is restarted.

For these experiments, we assume that the scheduler considers reservations when deciding which applications from the queue to start and that applications in the queue have priority over reservations when reservations are made. We performed simulations using only the ANL workload and 80 nodes due to time constraints.

Our first observation from our experiments is that varying the constant c between 0 and 600 has a very small impact on the number applications that are restarted. Therefore, the rest of the results we present assume that $c = 0$.

We vary the constants a and b and set c to zero to determine the optimal values for a and b. We choose a and b such that $a - b = 1.0$ and vary a between 0.0 and 1.0 in increments of 0.1. These values allow us to perform experiments varying the percentage of the termination cost associated with the amount of work performed from zero to one hundred percent with the amount of work yet to do contributing the remaining percent. The best values are shown in Tables 6.1 and 6.2. The tables show that the best values to use for the constants vary by the scheduling algorithm and if the mean wait time or mean difference from requested reservation time is being optimized. However, there are several trends that can be seen in the data. First, a is more positive than b is negative. This indicates that the amount of work done is the most important factor to consider when selecting which applications to terminate. Second, in over half of the cases both mean wait time and the mean difference in reservation is optimized with the same values of a and b. Third, from other simulations results, we observe that the mean wait times do not change smoothly as, say, a is increased from 0.0 to 1.0.

6.3.2 Considering Reservations While Scheduling

In this subsection we address the question of whether reservations should be considered when starting applications from a queue or not. If reservations are not considered, an application at the head of the queue would be started even if it would almost certainly have to be terminated to make nodes available for a reservation. If reservations are considered, the application at the head of the queue would

Table 6.1: Constants that minimize wait time when reservations are considered when starting queued applications.

Queue Ordering	Backfill	Percentage of Reservations	a	b	Mean Wait Time (minutes)	Mean Diff From Requested Reservation (minutes)
FCFS	no	10	0.6	-0.4	714.26	279.62
FCFS	no	20	0.8	-0.2	1109.97	445.14
FCFS	yes	10	0.8	-0.2	183.80	252.25
FCFS	yes	20	0.9	-0.1	194.30	331.45
LWF	no	10	0.6	-0.4	89.55	177.32
LWF	no	20	0.9	-0.1	97.80	242.57

Table 6.2: Constants that minimize the difference between reservation time and requested reservation time when reservations are considered when starting queued applications.

Queue Ordering	Backfill	Percentage of Reservations	a	b	Mean Wait Time (minutes)	Mean Diff From Requested Reservation (minutes)
FCFS	no	10	0.9	-0.1	722.09	265.76
FCFS	no	20	1.0	0.0	1110.11	399.96
FCFS	yes	10	0.8	-0.2	183.80	252.25
FCFS	yes	20	0.9	-0.1	194.30	331.45
LWF	no	10	0.6	-0.4	89.55	177.32
LWF	no	20	0.9	-0.1	97.80	242.57

Table 6.3: Constants that minimize wait time when reservations are not considered when starting queued applications.

Queue Ordering	Backfill	Percentage of Reservations	a	b	Mean Wait Time (minutes)	Mean Diff From Requested Reservation (minutes)
FCFS	no	10	0.9	-0.1	755.53	314.03
FCFS	no	20	0.9	-0.1	1109.16	389.50
FCFS	yes	10	0.8	-0.2	252.91	264.04
FCFS	yes	20	0.7	-0.3	480.09	370.49
LWF	no	10	1.0	0.0	111.11	181.97
LWF	no	20	0.9	-0.1	211.89	250.71

not be started, but perhaps an application later in the queue would be because it could execute before the nodes are needed by a reservation. Which technique to use depends on the accuracy of the run-time predictions. If the run-time predictions are not very accurate, the scheduler will start applications that have to be terminated and not start applications that could have executed to completion. These mistakes would increase the average wait time of applications.

Some of our simulation results are shown in Tables 6.3 and 6.4. If we compare this data to the data in the previous subsection, one can see that both mean wait time and the mean difference between reservation time and requested reservation time increase. The mean wait times increase by 22 percent on average and the mean offset from requested reservation time increases by only 1 percent. These results show that our run-time predictions are accurate enough that reservations should be considered when deciding which applications from the queue should be started.

Table 6.4: Constants that minimize the difference between reservation time and requested reservation time when reservations are not considered when starting queued applications.

Queue Ordering	Backfill	Percentage of Reservations	a	b	Mean Wait Time (minutes)	Mean Diff From Requested Reservation (minutes)
FCFS	no	10	1.0	0.0	770.65	263.56
FCFS	no	20	1.0	0.0	1165.79	382.78
FCFS	yes	10	0.8	-0.2	252.91	264.04
FCFS	yes	20	1.0	0.0	686.67	351.91
LWF	no	10	1.0	0.0	111.11	181.97
LWF	no	20	0.8	-0.2	238.98	242.88

6.3.3 Comparison to Nonrestartable Techniques

The previous two subsections present performance data for combining queuing scheduling and reservations when running applications can be terminated and restarted. We will now compare this performance to the scheduling performance when applications cannot be terminated and restarted. Table 6.5 presents the scheduling performance for simulating the ANL workload on 80 nodes when applications cannot be restarted and applications from the queue have priority over reservations when reservations are performed (the same assumption we make when applications can be restarted). Comparing this data to the data in Table 6.1 shows that if applications can be terminated and restarted, the mean wait time decreases by 8 percent and the mean difference from requested reservation time decreases by 124 percent. This shows that there is a performance benefit if we assume that applications are restartable, particularly in the mean difference from requested reservation time.

Table 6.5: Scheduling performance when applications cannot be terminated.

Queue Ordering	Backfilling	Percentage of Reservations	Mean Wait Time (minutes)	Mean Difference From Requested Reservation (minutes)
FCFS	no	10	650.55	570.08
FCFS	no	20	1348.20	1010.41
FCFS	yes	10	170.05	620.55
FCFS	yes	20	199.91	793.41
LWF	no	10	87.95	428.59
LWF	no	20	123.60	449.97

6.4 Related Work

Techniques for combining reservations from users with queuing scheduling is a relatively new area of research. User-level reservations are currently being added to the Portable Batch System (PBS) scheduler [57] and most scheduling systems allow administrators to assign nodes to certain users for certain times. The problem with this second technique is that it requires a human administrator in the process.

6.5 Summary

In this section we examine the performance of several different techniques for combining queuing scheduling with reservations. First, we examine techniques when applications cannot be restarted. We find that this forces us to use maximum run times for run-time predictions and techniques similar to backfilling. We find that forcing users to request reservations at least an hour or two ahead of time does not have any effect on scheduling performance, so using this constraint is a policy decision. We also find that supporting reservations does increase the wait times

of applications in the queue by 24 percent when 10 percent of the applications are reservations and by 144 percent when 20 percent of the applications are reservations. The large overhead when 20 percent of the applications are reservations is due to the SDSC workloads that have very large mean wait times. We show that if we decrease the percent of queued applications that cannot be delayed by a reservation from 100 to 50 then the mean wait time increases by an average of 8 percent and the mean difference from the requested reservation time decreases by 44 percent. If we decrease the percent of queued applications with priority from 100 to 0 percent then the mean wait time increases by 35 percent and the mean difference decreases by 97 percent.

Second, we evaluate scheduling techniques that assume that applications can be terminated and restarted at a later time. We use an equation to determine the cost of terminating each running application and use these costs when picking applications to terminate. We find that the cost should largely be determined by the amount of time the application has executed and the number of nodes it has used, but a prediction on the amount of time the execution has left to run should also be considered. Our data shows that reservations that have been made should be considered when determining which applications from the queue to start. Finally, if we assume that applications can be restarted and therefore run-time predictions other than maximum run-times can be used, the mean wait time is decreased by 8 percent on average and the mean difference between the requested reservation times and the actual reservation times decreases by 124 percent.

To summarize further, we find that in our environment, the best scheduling performance when combining queuing scheduling with reservations is achieved when:

- There are no constraints on when users can ask for reservations,

- backfilling of queued applications is supported,
- applications are restartable,
- run-time predictions more accurate then maximum run times are used, and
- reservations are considered when starting queued applications.

6.6 Future Work

There are several areas of future work that could be explored. First, more complicated techniques could be used to choose which applications to terminate when nodes are needed for reservations. Second, we could examine different scheduling algorithms. We saw in Chapter 5 that using more accurate run-time predictions in the scheduling algorithms we considered did not not always lead to better schedules. This same effect may occur here. Third, we could use the scheduling algorithm for applications in the queue when determining when a reservation can be made. We could allow reservations to be made on or after the time the application would run if it was submitted to the queue.

Chapter 7

Metacomputing Directory Service

High-performance distributed computing often requires careful selection and configuration of computers, networks, application protocols, and algorithms. These requirements do not arise in traditional distributed computing, where configuration problems can typically be avoided by the use of standard default protocols, interfaces, and so on. The situation is also quite different in traditional high-performance computing, where systems are usually homogeneous and hence can be configured manually. But in high-performance distributed computing, neither defaults nor manual configuration is acceptable. Defaults often do not result in acceptable performance, and manual configuration requires low-level knowledge of remote systems that an average programmer does not possess. We need an *information-rich* approach to configuration in which decisions are made (whether at compile-time, link-time, or run-time) based upon information about the structure and state of the system on which a program is to run.

An example from the I-WAY networking experiment illustrates some of the dif-

ficulties associated with the configuration of high-performance distributed systems. The I-WAY was composed of massively parallel computers, workstations, archival storage systems, and visualization devices [15]. These resources were interconnected by both the Internet and a dedicated 155 Mb/sec IP over ATM network. In this environment, applications might run on a single or multiple parallel computers, of the same or different types. An optimal communication configuration for a particular situation might use vendor-optimized communication protocols within a computer but TCP/IP between computers over an ATM network (if available). A significant amount of information must be available to select such configurations, for example:

- What are the network interfaces (i.e., IP addresses) for the ATM network and Internet?
- What is the raw bandwidth of the ATM network and the Internet, and which is higher?
- Is the ATM network currently available?
- Between which pairs of nodes can we use vendor protocols to access fast internal networks?
- Between which pairs of nodes must we use TCP/IP?

Additional information is required if we use a resource location service to select an "optimal" set of resources from among the machines available on the I-WAY at a given time.

In our experience, such configuration decisions are not difficult *if* the right information is available. Until now, however, this information has not been easily

available, and this lack of access has hindered application optimization. Furthermore, making this information available in a useful fashion is a nontrivial problem: the information required to configure high-performance distributed systems is diverse in scope, dynamic in value, distributed across the network, and detailed in nature.

In this chapter, we propose an approach to the design of high-performance distributed systems that addresses this need for efficient and scalable access to diverse, dynamic, and distributed information about the structure and state of resources. The core of this approach is the definition and implementation of a Metacomputing Directory Service (MDS) that provides a uniform interface to diverse information sources. We show how a simple data representation and application programming interface (API) based on the Lightweight Directory Access Protocol (LDAP) [40] meet requirements for uniformity, extensibility, and distributed maintenance. We introduce a data model suitable for distributed computing applications and show how this model is able to represent computers and networks of interest. We also present novel implementation techniques for this service that address the unique requirements of high-performance applications. Finally, we use examples from the Globus distributed computing toolkit to show how MDS data can be used to guide configuration decisions with realistic settings. We expect these techniques to be equally useful in other systems that support computing in distributed environments, such as Legion [34], NEOS [14], NetSolve [10], Condor [46], Nimrod [1], PRM [50], AppLeS [4], and heterogeneous implementations of MPI [24].

The principal contributions of this chapter are

- a new architecture for high-performance distributed computing systems, based upon an information service called the Metacomputing Directory Service;

- a design for this directory service, addressing issues of data representation, data model, and implementation;
- a data model able to represent the network structures commonly used by distributed computing systems, including various types of supercomputers; and
- a demonstration of the use of the information provided by MDS to guide resource and communication configuration within a distributed computing toolkit.

I helped to design the directory service and define the data model.

The rest of this chapter is organized as follows. In Section 7.1, we explain the requirements that a distributed computing information infrastructure must satisfy, and we propose MDS in response to these requirements. We then describe the representation (Section 7.2), the data model (Section 7.3), and the implementation (Section 7.4) of MDS. In Section 7.5, we demonstrate how MDS information is used within Globus. We conclude in Section 7.6 with suggestions for future research efforts.

7.1 Designing a Metacomputing Directory Service

The problem of organizing and providing access to information is a familiar one in computer science, and there are many potential approaches to the problem, ranging from database systems to the Simple Network Management Protocol (SNMP). The

appropriate solution depends on the ways in which the information is produced, maintained, accessed, and used.

7.1.1 Requirements

Following are the requirements that shaped our design of an information infrastructure for distributed computing applications. Some of these requirements can be expressed in quantitative terms (e.g., scalability, performance); others are more subjective (e.g., expressiveness, deployability).

Performance. The applications of interest to us frequently operate on a large scale (e.g., hundreds of processors) and have demanding performance requirements. Hence, an information infrastructure must permit rapid access to frequently used configuration information. It is not acceptable to contact a server for every item: caching is required.

Scalability and cost. The infrastructure must scale to large numbers of components and permit concurrent access by many entities. At the same time, its organization must permit easy discovery of information. The human and resource costs (CPU cycles, disk space, network bandwidth) of creating and maintaining information must also be low, both at individual sites and in total.

Uniformity. Our goal is to simplify the development of tools and applications that use data to guide configuration decisions. We require a uniform data model as well as an application programming interface (API) for common operations on the data represented via that model. One aspect of this uniformity is a

standard representation for data about common resources, such as processors and networks.

Expressiveness. We require a data model rich enough to represent relevant structure within distributed computing systems. A particular challenge is representing characteristics that span organizations, for example network bandwidth between sites.

Extensibility. Any data model that we define will be incomplete. Hence, the ability to incorporate additional information is important. For example, an application can use this facility to record specific information about its behavior (observed bandwidth, memory requirements) for use in subsequent runs.

Multiple information sources. The information that we require may be generated by many different sources. Consequently, an information infrastructure must integrate information from multiple sources.

Dynamic data. Some of the data required by applications is highly dynamic: for example, network availability or load. An information infrastructure must be able to make this data available in a timely fashion.

Flexible access. We require the ability to both read and update data contained within the information infrastructure. Some form of search capability is also required, to assist in locating stored data.

Security. It is important to control who is allowed to update configuration data. Some sites will also want to control access.

Deployability. An information infrastructure is useful only if is broadly deployed.

In the current case, we require techniques that can be installed and maintained easily at many sites.

Decentralized maintenance. It must be possible to delegate the task of creating and maintaining information about resources to the sites at which resources are located. This delegation is important for both scalability and security reasons.

7.1.2 A Metacomputing Directory Service

Our analysis of requirements and existing systems leads us to define what we call the Metacomputing Directory Service (MDS). This system consists of three distinct components:

1. **Representation and data access**: The directory structure, data representation, and API defined by LDAP.

2. **Data model**: A data model that is able to encode the types of resources found in high-performance distributed computing systems.

3. **Implementation**: A set of implementation strategies designed to meet requirements for performance, multiple data sources, and scalability.

We provide more details on each of these components in the following sections.

Figure 7.1 illustrates the structure of MDS and its role in a high-performance distributed computing system. An application running in a distributed computing environment can access information about system structure and state through a uniform API. This information is obtained through the MDS client library, which may access a variety of services and data sources when servicing a query.

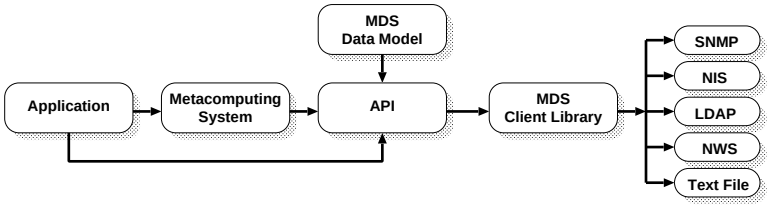

Figure 7.1: Overview of the architecture of the Metacomputing Directory Service.

7.2 Representation

The MDS design adopts the data representations and API defined by the LDAP directory service. This choice is driven by several considerations. Not only is the LDAP data representation extensible and flexible, but LDAP is beginning to play a significant role in Web-based systems. Hence, we can expect wide deployment of LDAP information services, familiarity with LDAP data formats and programming, and the existence of LDAP directories with useful information. Note that the use of LDAP representations and API does not constrain us to use standard LDAP implementations. As we explain in Section 7.4, the requirements of high-performance distributed computing applications require alternative implementation techniques. However, LDAP provides an attractive interface on which we can base our implementation. LDAP also provides a mechanism to restrict the types of operations that can be performed on data, which helps to address our security requirements.

In the rest of this section, we talk about the "MDS representation," although this representation comes directly from LDAP (which in turn "borrows" its representation from X.500 [47]). In this representation, related information is organized into well-defined collections, called entries. MDS contains many entries, each representing an instance of some type of object, such as an organization, person, network, or

computer. Information about an entry is represented by one or more attributes, each consisting of a name and a corresponding value. The attributes that are associated with a particular entry are determined by the type of object the entry represents. This type information, which is encoded within the MDS data model, is encoded in MDS by associating an *object class* with each entry. We now describe how entries are named and then, how attributes are associated with objects.

7.2.1 Naming MDS Entries

Each MDS entry is identified by a unique name, called its *distinguished name*. To simplify the process of locating an MDS entry, entries are organized to form a hierarchical, tree-structured name space called a *directory information tree* (DIT). The distinguished name for an entry is constructed by specifying the entries on the path from the DIT root to the entry being named.

Each component of the path that forms the distinguished name must identify a specific DIT entry. To enable this, we require that, for any DIT entry, the children of that entry must have at least one attribute, specified a priori, whose value distinguishes it from its siblings. (The X.500 representation actually allows more than one attribute to be used to disambiguate names.) Any entry can then be uniquely named by the list of attribute names and values that identify its ancestors up to the root of the DIT. For example, consider the following MDS distinguished name:

```
< hn = dark.mcs.anl.gov,
  ou = MCS,
  o  = Argonne National Laboratory,
  o  = Globus,
  c  = US >
```

The components of the distinguished name are listed in *little endian* order, with the component corresponding to the root of the DIT listed last. Within a distinguished name, abbreviated attribute names are typically used. Thus, in this example, the names of the distinguishing attributes are: host name (hn), organizational unit (ou), organization (o), and country (c). Thus, a country entry is at the root of the DIT, while host entries are located beneath the organizational unit level of the DIT (see Figure 7.2). In addition to the conventional set of country and organizational entries (US, ANL, USC, etc.), we incorporate an entry for a pseudo-organization named "Globus," so that the distinguished names that we define do not clash with those defined for other purposes.

7.2.2 Object Classes

Each DIT entry has a user-defined type, called its *object class*. (LDAP defines a set of standard object class definitions, which can be extended for a particular site.) The object class of an entry defines which attributes are associated with that entry and what type of values those attributes may contain. For example, Figure 7.3 shows the definition of the object classes `GlobusHost` and `GlobusResource`, and Figure 7.4 shows the values associated with a particular host. The object class definition consists of three parts: a parent class, a list of required attributes, and a

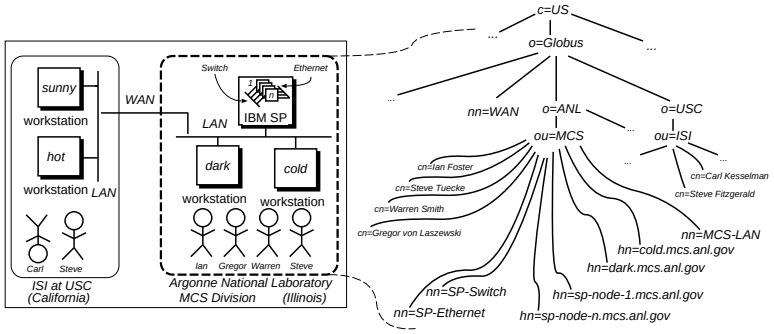

Figure 7.2: A subset of the DIT defined by MDS, showing the organizational nodes for Globus, ANL, and USC; the organizational units ISI and MCS; and a number of people, hosts, and networks.

list of optional attributes.

The `SUBCLASS` section of the object class definition enables a simple inheritance mechanism, allowing an object class to be defined in terms of an extension of an existing object class. The `MUST CONTAIN` and `MAY CONTAIN` sections specify the required and optional attributes found in an entry of this object class. Following each attribute name is the type of the attribute value. While the set of attribute types is extensible, a core set has been defined, including case-insensitive strings (`cis`) and distinguished names (`dn`).

In Figure 7.3, `GlobusHost` inherits from the object class `GlobusResource`. This means that a `GlobusHost` entry (i.e., an entry of type `GlobusHost`) contains all of the attributes required by the `GlobusResource` class, as well as the attributes defined within its own `MUST CONTAIN` section. In Figure 7.4, the administrator attribute is inherited from `GlobusResource`. A `GlobusHost` entry may also optionally contain the attributes from both its parent's and its own `MAY CONTAIN` section.

Notice that the administrator attribute in Figure 7.4 contains a distinguished

```
GlobusHost OBJECT CLASS
  SUBCLASS OF GlobusResource
  MUST CONTAIN {
    hostName          :: cis,
    type              :: cis,
    vendor            :: cis,
    model             :: cis,
    OStype            :: cis,
    OSversion         :: cis
  }
  MAY CONTAIN {
    networkNode        :: dn,
    totalMemory        :: cis,
    totalSwap          :: cis,
    dataCache          :: cis,
    instructionCache   :: cis
  }
```

```
GlobusResource OBJECT CLASS
  SUBCLASS OF GlobusTop
  MUST CONTAIN {
    administrator :: dn
  }
  MAY CONTAIN {
    manager       :: dn,
    provider      :: dn,
    technician    :: dn,
    description   :: cis,
    documentation :: cis
  }
```

Figure 7.3: Simplified versions of the MDS object classes `GlobusHost` and `GlobusResource`.

name. This distinguished name acts as a pointer, linking the host entry to the person entry representing the administrator. One must be careful not to confuse this link, which is part of an entry, with the relationships represented by the DIT, which are not entry attributes. The DIT should be thought of as a separate structure used to organize an arbitrary collection of entries and, in particular, to enable the distribution of these entries over multiple physical sites. Using distinguished names as attribute values enables one to construct more complex relationships than the trees found in the DIT. The ability to define more complex structures is essential for our purposes, since many distributed computing structures are most naturally represented as graphs.

```
dn: hn=dark.mcs.anl.gov, ou=MCS,
    o=Argonne National Laboratory, o=Globus, c=US
objectclass:   GlobusHost
objectclass:   GlobusResource
administrator: cn=John Smith, ou=MCS,
               o=Argonne National Laboratory, o=Globus, c=US
hostName:      dark.mcs.anl.gov
type:          sparc
vendor:        Sun
model:         SPARCstation-10
OStype:        SunOS
OSversion:     5.5.1
```

Figure 7.4: Sample data representation for an MDS computer

7.3 Data Model

To use the MDS representation for a particular purpose, we must define a data model in which information of interest can be maintained. This data model must specify both a DIT hierarchy and the object classes used to define each type of entry.

In its upper levels, the DIT used by MDS (see Figure 7.2) is typical for LDAP directory structures, looking similar to the organization used for multinational corporations. The root node is of object class *country*, under which we place first the *organization* entry representing Globus and then the *organization* and *organizational unit* (i.e., division or department) entries. Entries representing people and computers are placed under the appropriate organizational units.

The representation of computers and networks is central to the effective use of MDS, and so we focus on this issue in this section.

7.3.1 Representing Networks and Computers

We adopt the framework for representing networks introduced in RFC 1609 [47] as the starting point for the representation used in MDS. However, the RFC 1609 framework provides a network-centric view in which computers are accessible only via the networks to which they are connected. We require a representation of networks and computers that allows us to answer questions such as

- Are computers A and B on the same local area network?
- What is the latency between computers C and D?
- What protocols are available to communicate between computers E and F?

In answering these questions, we often require access to information about networks, but questions are posed most often from the perspective of the computational resource. That is, they are computer-centric questions. Our data model reflects this perspective.

A high-level view of the DIT structure used in MDS is shown in Figure 7.2. As indicated in this figure, both people and hosts are immediate children of the organizations in which they are located. For example, the distinguished name

```
< hn=dark.mcs.anl.gov,
  ou=MCS, o=Argonne National Laboratory,
  o=Globus, c=US >
```

identifies a computer administered by the Mathematics and Computer Science (MCS) Division at Argonne National Laboratory.

Communication networks are also explicitly represented in the DIT as children of an organization. For example, the distinguished name

```
< nn=mcs-lan,
  ou=MCS, o=Argonne National Laboratory,
  o=Globus, c=US >
```

represents the local area network managed by MCS. This distinguished name identifies an instance of a `GlobusNetwork` object. The attribute values of a `GlobusNetwork` object provides information about the *physical* network link, such as the link protocol (e.g., ATM or Ethernet), network topology (e.g., bus or ring type), and physical media (e.g., copper or fiber). As we shall soon see, logical information, such as the network protocol being used, is *not* specified in the `GlobusNetwork` object but is associated with a `GlobusNetworkImage` object. Networks that span organizations can be represented by placing the `GlobusNetwork` object higher in the DIT.

Networks and hosts are related to one another via `GlobusNetworkInterface` objects: hosts contain network interfaces, and network interfaces are attached to networks. A network interface object represents the physical characteristics of a network interface (such as interface speed) and the hardware network address (e.g. the 48-bit Ethernet address in the case of Ethernet). Network interfaces appear under hosts in the DIT, while a network interface is associated with a network via an attribute whose value is a distinguished name pointing to a `GlobusNetwork` object. A reverse link exists from the `GlobusNetwork` object back to the interface.

To illustrate the relationship between `GlobusHost`, `GlobusNetwork`, and `GlobusNetworkInterface` objects, we consider the configuration shown in Figure 7.5. This configuration consists of an IBM SP parallel computer and two workstations, all associated with MCS. The SP has two networks: an internal high-speed switch and an Ethernet; the workstations are connected only to an Ethernet. Although the

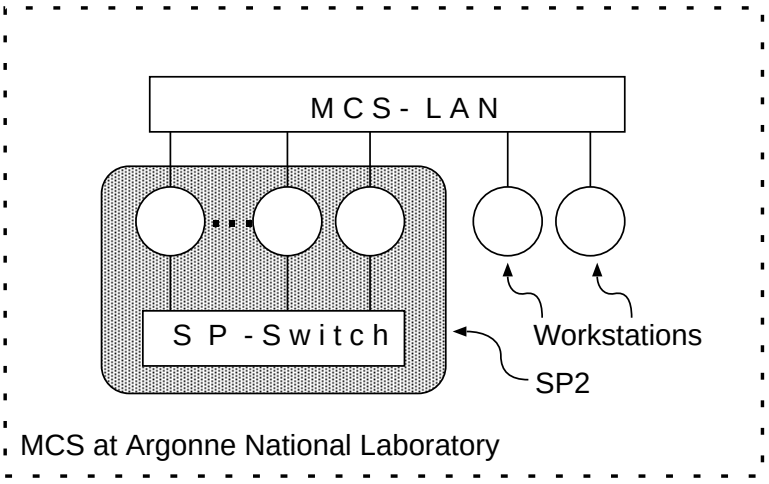

Figure 7.5: A configuration comprising two networks and $N+2$ computers.

SP Ethernet and the workstation Ethernet are connected via a router, we choose to represent them as a single network. An alternative, higher-fidelity MDS representation would capture the fact that there are two interconnected Ethernet networks.

The MDS representation for Figure 7.5 is shown in Figure 7.6. Each host and network in the configuration appear in the DIT directly under the entry representing MCS at Argonne National Laboratory. Note that individual SP nodes are children of MCS. This somewhat unexpected representation is a consequence of the SP architecture: each node is a fully featured workstation, potentially allowing login. Thus, the MDS representation captures the dual nature of the SP as a parallel computer (via the switch network object) and as a collection of workstations.

As discussed above, the `GlobusNetworkInterface` objects are located in the DIT under the `GlobusHost` objects. Note that a `GlobusHost` can have more than one network interface entry below it. Each entry corresponds to a different physical network connection. In the case of an SP, each node has at least two network

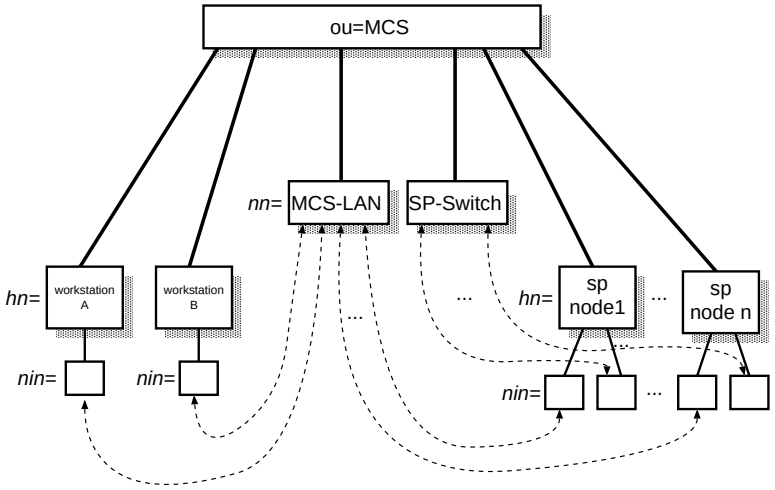

Figure 7.6: The MDS representation of the configuration depicted in Figure 7.5, showing host (HN), network (NN), and network interface (NIN) objects. The dashed lines correspond to "pointers" represented by distinguished name attributes.

interfaces: one to the high-speed switch and one to an Ethernet. Finally, we see that distinguished names are used to complete the representation, linking the network interface and network object together.

7.3.2 Logical Views and Images

At this point, we have described the representation of a physical network: essentially link-level aspects of the network and characteristics of network interface cards and the hosts they plug into. However, a physical network may support several "logical" views, and we may need to associate additional information with these logical views. For example, a single network might be accessible via several different protocol stacks: IP, Novell IPX, or vendor-provided libraries such as MPI. Associated with each of these protocols can be distinct network interface and performance information. Additionally, a "partition" might be created containing a subset of

available computers; scheduling information can be associated with this object.

The RFC 1609 framework introduces the valuable concept of *images* as a mechanism for representing multiple logical views of the same physical network. We apply the same concept in our data model. Where physical networks are represented by `GlobusHost`, `GlobusNetwork`, and `GlobusNetworkInterface` object classes, network images are represented by `GlobusHostImage`, `GlobusNetworkImage`, and `GlobusNetworkInterfaceImage` object classes. Each image object class contains new information associated with the logical view, as well as a distinguished name pointing to its relevant physical object. In addition, a physical object has distinguished name pointers to all of the images that refer to it. For example, one may use both IP and IPX protocols over a single Ethernet interface card. We would represent this in MDS by creating two `GlobusNetworkInterfaceImage` objects. One image object would represent the IP network and contain the IP address of the interface, as well as a pointer back to the object class representing the Ethernet card. The second image object would contain the IPX address, as well as a distinguished name pointing back to the same entry for the Ethernet card. The `GlobusNetworkInterface` object would include the distinguished names of both interface images.

The structure of network images parallels that of the corresponding physical networks, with the exception that not all network interfaces attached to a host need appear in an image. To see why, consider the case of the IBM SP. One might construct a network image to represent the "parallel computer" view of the machine in which IBM's proprietary message-passing library is used for communication. Since this protocol cannot be used over the Ethernet, this image of the network will not contain images representing the Ethernet card. Note that we can also produce a network image of the SP representing the use of IP protocols. This view may include

images of both the switch and Ethernet network interfaces.

7.3.3 Questions Revisited

At this stage we have gone quite deeply into the representation of computers and networks but have strayed rather far from the issue that motivated the MDS design, namely, the configuration of high-performance distributed computations. To see how MDS information can be used, let us revisit the questions posed at the start of this chapter with respect to the use of multiple computers on the I-WAY:

- *What are the network interfaces (i.e., IP addresses) for the ATM network and Internet?* A host's IP address on the ATM network can be found by looking for a `GlobusNetworkInterface` that is pointing to a `GlobusNetwork` with a link protocol attribute value of ATM. From the interface, we find the `GlobusNetworkInterfaceImage` representing an IP network, and the IP address will be stored as an attribute in this object.

- *What is the raw bandwidth of the ATM network and the Internet, and which is higher? Is the ATM network currently available?* The raw bandwidth of the ATM network will be stored in the I-WAY `GlobusNetwork` object. Information about the availability of the ATM network can also be maintained in this object.

- *Between which pairs of nodes can we use vendor protocols to access fast internal networks? Between which pairs of nodes must we use TCP/IP?* Two nodes can communicate using a vendor protocol if they both point to `GlobusHostImage` objects that belong to the same `GlobusNetworkImage` object.

Note that the definition of the MDS representation, API, and data model means that this information can be obtained via a single mechanism, regardless of the computers on which an application actually runs.

7.4 Implementation

We have discussed how information is represented in MDS, and we have shown how this information can be used to answer questions about system configuration. We now turn our attention to the MDS implementation. Since our data model has been defined completely within the LDAP framework, we could in principle adopt the standard LDAP implementation. This implementation uses a TCP-based wire protocol and a distributed collection of servers, where each server is responsible for all the entries located within a complete subtree of the DIT. While this approach is suitable for a loosely coupled, distributed environment, it has three significant drawbacks in a high-performance environment:

- **Single information provider**. The LDAP implementation assumes that all information within a DIT subtree is provided by a single information provider. (While some LDAP servers allow alternative "backend" mechanisms for storing entries, the same backend must be used for all entries in the DIT subtree.) However, restricting all attributes to the same information provider complicates the design of the MDS data-model. For example, the IP address associated with a network interface image can be provided by a system call, while the network bandwidth available through that interface is provided by a service such as the Network Weather Service (NWS) [62].

- **Client/server architecture**. The LDAP implementation requires at least one round-trip network communication for each LDAP access. Frequent MDS accesses thus becomes prohibitively expensive. We need a mechanism by which MDS data can be cached locally for a timely response.

- **Scope of Data**. The LDAP implementation assumes that any piece of information may be used from any point in the network (within the constraints of access control). However, a more efficient implementation of attribute update can be obtained if one can limit the locations from which attribute values can be accessed. The introduction of scope helps to determine which information must be propagated to which information providers, and when information can be safely cached.

Note that these drawbacks all relate to the LDAP implementation, not its API. Indeed, we can adopt the LDAP API for MDS without modification. Furthermore, for those DIT subtrees that contain information that is not adversely affected by the above limitations, we can pass the API calls straight through to an existing LDAP implementation. In general, however, MDS needs a specialized implementation of the LDAP API to meet the requirements for high performance and multiple information providers.

The most basic difference between our MDS implementation and standard LDAP implementations is that we allow information providers to be specified on a *per attribute* basis. Referring to the above example, we can provide the IP address of an interface via SNMP, the current available bandwidth via NWS, and the name of the machine into which the interface card is connected. Additionally, these providers can store information into MDS on a periodic basis, thus allowing refreshing of dynamic

information. The specification of which protocol to use for each entry attribute is stored in an *object class metadata entry*. Metadata entries are stored in MDS and accessed via the LDAP protocol.

In addition to specifying the access protocol for an attribute, the MDS object class metadata also contains a time-to-live (TTL) for attribute values and the update scope of the attribute. The TTL data is used to enable caching; a TTL of 0 indicates that the attribute value cannot be cached, while a TTL of -1 indicates that the data is constant. Positive TTL values determine the amount of time that the attribute value is allowed to be provided out of the cache before refreshing.

The update scope of an attribute limits the readers of an updated attribute value. Our initial implementation considers three update scopes: process, computation, and global. Process scope attributes are accessible only within the same process as the writer, whereas computation scope attributes can be accessed by any process within a single computation, and global scope attributes can be accessed from any node or process on a network.

7.5 MDS Applications in Globus

We review briefly some of the ways in which MDS information can be used in high-performance distributed computing. We focus on applications within Globus. These mechanisms include communication, authentication, resource location, resource allocation, process management, and (in the form of MDS) information infrastructure.

The Globus toolkit is designed with the configuration problem in mind. It attempts to provide, for each of its components, interfaces that allow higher-level services to manage how low-level mechanisms are applied. As an example, we consider

the problem referred to earlier of selecting network interfaces and communication protocols when executing communication code within a heterogeneous network. The Globus communication module (a library called Nexus) allows a user to specify an application's communication operations by using a single notation, regardless of the target platform: either the Nexus API or some library or language layered on top of that API. At run-time, the Nexus implementation configures a communication structure for the application, selecting for each communication link (a Nexus construct) the communication method that is to be used for communications over that link. In making this selection for a particular pair of processors, Nexus first uses MDS information to determine which low-level mechanisms are available between the processors. Then, it selects from among these mechanisms, currently on the basis of built-in rules (e.g., "ATM is better than Internet"); rules based on dynamic information ("use ATM if current load is low"), or programmer-specified preferences ("always use Internet because I believe it is more reliable") can also be supported in principle. The result is that application source code can run unchanged in many different environments, selecting appropriate mechanisms in each case.

These method-selection mechanisms were used in the I-WAY testbed to permit applications to run on diverse heterogeneous virtual machines [28]. For example, on a virtual machine connecting IBM SP and SGI Challenge computers with both ATM and Internet networks, Nexus used three different protocols (IBM proprietary MPL on the SP, shared-memory on the Challenge, and TCP/IP or AAL5 between computers) and selected either ATM or Internet network interfaces, depending on network status.

7.5.1 Related Work

Harvest [6] is a tool for gathering information from Internet repositories, building topic-specific indexes, and searching the indexes. Harvest consists of providers, gatherers, brokers, caches, replicators, and the harvest registry service. Providers are entities such as http and ftp servers that provide information. Gatherers collect and extract indexing information from providers. Ideally, a gatherer will run on the same host as a provider. The gatherer then only sends summary information to brokers, reducing network traffic. Brokers retrieve information from gatherers and other brokers to provide indexing and a query interface to gathered information. The brokers produce an index for the information it received from gatherers and providers and performs searches using this index in response to user queries. A hierarchical caching system is available to maintain copies of objects from providers. These caches are hopefully faster to access then the provider. The replicators maintain replicas of other components. This results in faster accesses to each component. A final component is the harvest server registry that allows users to register information about each gatherer, broker, cache, and replicator. The registry is used for finding appropriate brokers and constructing new brokers and gatherers so that their is no duplication of effort.

The CORBA trading service [2] allows the selection of service providers at run time. This is accomplished by service providers exporting a description of the services they provide to a trader and then allowing an application to import a service by asking a trader for an appropriate service provider. A service is identified by an interface to which a connection can be established, a type, and a set of properties describing the service. A service request contains the service type, constraints on the properties for the service, preferences used to order service offers, policies that

describe how the search should be performed, and the service properties that should be returned from the query.

The traders accept service offers from exporters of services and service requests from importers of services. Traders attempt to match the service request to one or more service offers. Traders can be linked together so that service request can be passed to other traders until it can be satisfied or the search fails. Traders and the links between traders have characteristics that determine how a search is performed. These characteristics include the maximum number of traders to query for each request, rules on which links to follow, etc.

X.500 [47] is a standardized directory system. The directory information is maintained in Directory Service Agents (DSAs) and is organized in a Directory Information Tree (DIT), similar to the domain name system. The top DSA is the root, the next level consist of country DSAs, then organizations, and so on. Each DSA holds part of the global directory and is able to find the DSA that holds any part of the global directory by traversing the tree. Information is stored in entries that consist of attributes and associated values. Any subtree of the directory can be searched by specifying an attribute and value. This structure results in inefficient global searches since all DSAs must be queried. X.500 defines a Directory Access Protocol (DAP) to access DSAs. This protocol is based over the OSI network stack and requires significant computational resources to use.

The Lightweight Directory Access Protocol (LDAP) [63, 39, 40] is designed to allow access to X.500 directories without the overhead of DAP. LDAP has lower overhead than DAP because requests are carried over TCP/IP bypassing session/transport overhead, many data elements are encoded as ordinary strings, and uses a lightweight encoding for all protocol elements. The most commonly used operations

provided by LDAP are a bind operation for a client to connect to a LDAP server, a search operation to search for entries, a modify operation to modify the directory, and operations to add and delete elements. The search operation can specify many search characteristics such as the subtree to search, how long to search, the maximum number of entries to find, the attributes to be returned for each entry, and the filter [39] to be used to determine if an entry matches a query.

Whois++ [60] is similar to X.500 in that it represents a directory of entries (called records) which consist of attributes and values. The Whois++ directory model does not define a hierarchical name structure like X.500. Whois++ has a set of index servers and Whois++ servers. The Whois++ servers contain the records and each one is indexed by at least one index server. The index servers contain centroids for Whois++ servers. A centroid consists of a list of the values that occur for each attribute in any record in the Whois++ server. Index servers can be indexed by other servers to produce a hierarchy where higher level index servers have a centroid for each index server it indexes.

Clients find records in this structure by first contacting an appropriate index server. An appropriate index server is usually the physically closest server to the client. The client then sends a search request and the response is either records that satisfy the search (if the query went to a Whois++ server), other index servers to contact, or that no information was found. In the first case, the search is complete. In the second case, the client can contact the specified index servers to continue the search for Whois++ servers. In the third case, the client asks the index server for the names of index servers that index it. The search can then proceed through these higher-level servers. Whois++ is more efficient than X.500 for searching but is less efficient for browsing since searches must be used to build lists of data.

Reviewing these various systems, we see that each is in some way incomplete, failing to address the types of information needed to build high-performance distributed computing systems, being too slow, or not defining an API to enable uniform access to the service. For these reasons, we have defined our own metacomputing information infrastructure that integrates existing systems while providing a uniform and extensible data model, support for multiple information service providers, and a uniform API.

7.6 Summary

We have argued that the complex, heterogeneous, and dynamic nature of high-performance distributed computing systems requires an *information-rich* approach to system configuration. In this approach, tools and applications do not rely on defaults or programmer-supplied knowledge to make configuration choices. Instead, they base choices on information obtained from external sources.

With the goal of enabling information-rich configuration, we have designed and implemented a *Metacomputing Directory Service*. MDS is designed to provide uniform, efficient, and scalable access to dynamic, distributed, and diverse information about the structure and state of resources. MDS defines a representation (based on that of LDAP), a data model (capable of representing various parallel computers and networks), and an implementation (which uses caching and other strategies to meet performance requirements). Experiments conducted with the Globus toolkit (particularly in the context of the I-WAY) show that MDS information can be used to good effect in practical situations.

7.7 Future Work

The MDS has been deployed in our GUSTO distributed computing testbed and we are extending additional Globus components to use MDS information for configuration purposes. The MDS is being adopted as a standard information service for metacomputing systems and the data representations are changing slightly to accommodate this.

Other directions for immediate investigation include expanding the set of information sources supported, evaluating performance issues in applications, and developing optimized implementations for common operations. In the longer term, we are interested in more sophisticated applications (e.g., source routing, resource scheduling) and in the recording and use of application-generated performance metrics. Another application for MDS information that we are investigating is resource location [59]. A "resource broker" is basically a process that supports specialized searches against MDS information. Rather than incorporate these search capabilities in MDS servers, we plan to construct resource brokers that construct and maintain the necessary indexes, querying MDS periodically to obtain up-to-date information. We are also investigating techniques for replicating and distributing the data in the information service. The goal of this effort is to increase both the performance and availability of the information service. Finally, we are investigating support for referral to other information services. A referral service allows our information service to retrieve requested information from another information service.

Chapter 8

Resource Management Interface

This chapter describes a common interface to local resource managers. An interface such as this is needed by users of metacomputing systems because there are many different local scheduling systems in use, as well as computers that do not have schedulers at all. This common interface allows users to understand only a single interface and be able to execute applications on many different computer systems. This interface is used in Globus and is called the Globus Resource Allocation Manager (GRAM). A GRAM allows a remote user to start, monitor, and terminate applications on a computer system. The GRAMs provide a single interface to for managing applications on many types of computer systems. Some examples are single workstations, workstations scheduled by systems such as Condor [46], and parallel computers scheduled by scheduling systems such as LoadLeveler, EASY [44], Maui, NQE, LSF [51], and PBS [57].

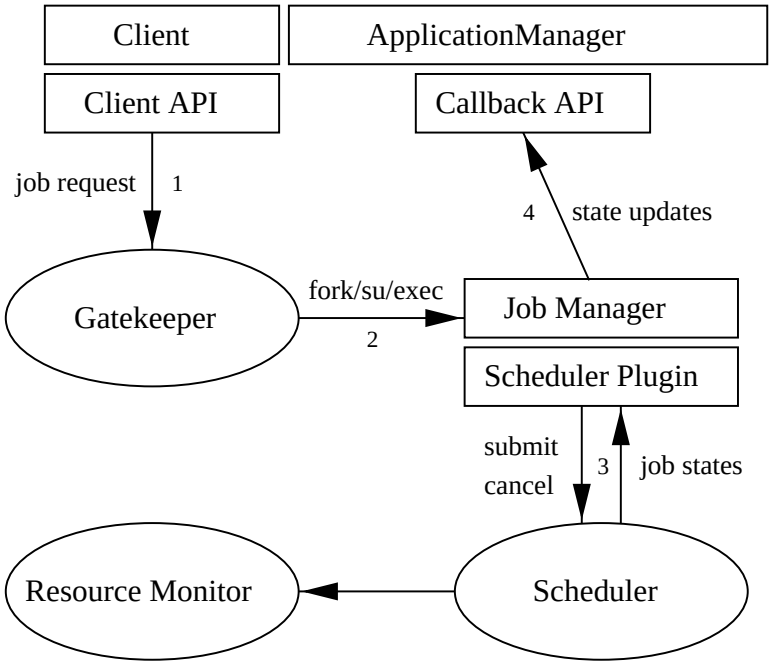

Figure 8.1: GRAM Architecture.

8.1 Architecture

The GRAM architecture is shown in Figure 8.1. To execute an application through a GRAM, a user submits an application through a GRAM client. A GRAM client can either be written by the user using the GRAM client API or there are several GRAM clients provided with Globus that are used as command line tools. The GRAM client forwards the submission to a gatekeeper on the remote computer. The gatekeeper authenticates the user, creates a job manager running under the user's local user ID, and passes information about the application to the job manager. The job manager then starts and monitors the application, communicating state changes back to the GRAM client that started the application. When the remote application terminates, normally or by failing, the job manager terminates as well. The enties in our system

are:

Resource - an entity capable of running one or more processes on behalf of a user.

Client - the process that is using the resource allocation client-side API.

Job - a process or set of processes resulting from a job request. Jobs are grouped, so any error in one job results in the mutual termination of all others in the group. If the job is killed by the client, all processes are terminated, and the job itself is finally terminated as well.

Job Request - a request to gatekeeper to create one or more job processes, expressed in the supplied Resource Specification Language. This request guides

- resource selection (when and where to create the job processes)
- job process creation (what job processes to create)
- job control (how the processes should execute)

Gatekeeper - a process, running as root, which begins the process of handling allocation requests. It exists on the remote computer before any request is submitted. When the gatekeeper receives an allocation request from a client, it

- mutually authenticates with the client,
- maps the requester to a local user,
- starts a job manager on the local host as the local user, and
- passes the allocation arguments to the newly created job manager

Job Manager - one job manager is created by the gatekeeper to fulfill every request submitted to the gatekeeper. It does this by fork/exec or by communicating with a scheduling system. It starts the job on the local system, and handles all further communication with the client. It is made up of two components:

- *Common Component* - translates messages received from the gatekeeper and client into an internal API that is implemented by the machine specific component. It also translates callback requests from the machine specific component through the internal API into messages to the application manager.
- *Machine-Specific Component* - implements the internal API in the local environment. This includes calls to the local system, messages to the resource monitor, and inquiries to the MDS.

Resource Monitor - monitors the local scheduling system and resources, providing estimated delay times to the job manager, as well as others.

Application Manager - manages applications running on a variety of resources through many resource managers. As the central manager, it receives the callback requests from the job manager, indicating a change in job status.

8.2 Scheduling Model

Our resource management interface support the following scheduling model. A user (or resource broker acting on behalf of the user) submits a job request to a resource manager. The user or broker may have previously used MDS inquiry functions to identify resource managers that have resources that meet user requirements. When

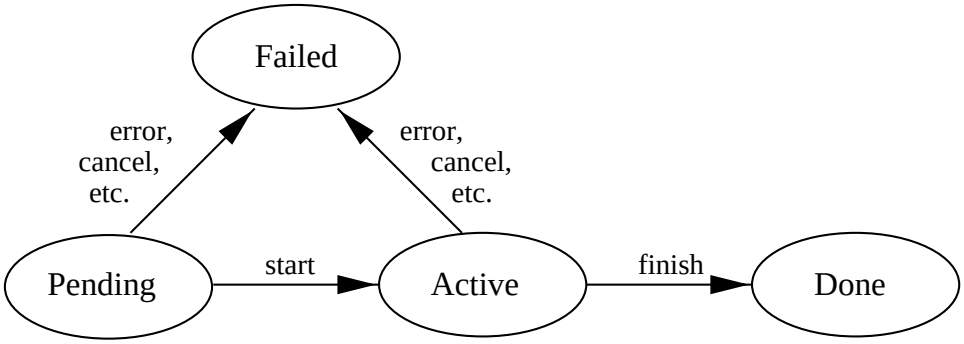

Figure 8.2: GRAM job state transitions.

submitted, the job is initially pending. The job may then undergo the state transitions illustrated in Figure 8.2.

The possible job states are:

- Pending: Resources have not yet been allocated to the job.
- Failed: The user job terminated before the job processes completed successfully. This can occur when an error occurs in a job process or when the user or system cancels the job.
- Active: The job has received resources and the application is running.
- Done: The job ran successfully.

We include a "Failed" state as well as having a submission request to return an error code, because on some systems a request may not be determined to have failed until after it is submitted.

The nature and timing of the transition from Pending to Active depends on the local scheduler: the job request may be handled immediately, enqueued, or

deferred until a user-specified time. Users can query MDS to determine how the local scheduler handles requests.

The job states are encoded in the following constants, used in the `gram_callback_allow()` and `gram_callback_check()` functions described below. These states can be combined using a bitwise OR operation.

- GRAM_JOB_STATE_PENDING
- GRAM_JOB_STATE_FAILED
- GRAM_JOB_STATE_ACTIVE
- GRAM_JOB_STATE_DONE

There are a large number error codes that are used to indicate why a job failed.

8.3 Resource Specification Language

The Globus RSL is a language that is used to describe the resources required for an application. The language is based on the LDAP [40] query language and uses relations of the form *(attribute op value)* and logical operators to describe the resources the application wants to use and how the application wants to use the resources. A simple example is shown below.

```
&(resourceManagerContact="denali.mcs.anl.gov:8713:/C=US/
    O=Globus/O=Argonne National Laboratory/OU=Mathematics
    and Computer Science Division/CN=denali.mcs.anl.gov-fork")
 (count=16)
 (executable="/sandbox/wsmith/cactus-3.2/cactus_irix6")
```

```
(arguments="/home/wsmith/cactus/neutron-128-128.par")
```

This request specifies the GRAM to contact (resourceManagerContact), the number of processes to start up (count), the executable to run (executable), and the arguments to pass to the executable (arguments). The ampersand symbol is a logical and operator. The operator comes before the operands and operates on two or more operands.

The RSL also contains the OR ("|") logical operator. This operator is not supported by lower-level Globus tools such as GRAMs, but it useful for specifying alternatives in a higher-level tool. For example, a user could ask for 16 nodes of a SGI Origin or 32 nodes of a Cray T3E using:

```
|(&(osname="irix")(count=16)
   (executable="ftp://tuva.mcs.anl.gov/sandbox/wsmith/
      cactus-cactus-3.2/cactus_irix6"))
   (arguments ="ftp://tuva.mcs.anl.gov/home/wsmith/
      cactus/neutron-128-128.par")
 (&(osname="unicos")(count=32)
   (executable="ftp://tuva.mcs.anl.gov/sandbox/wsmith/
      cactus-cactus-3.2/cactus_t3e"))
```

The URLs above beginning with "ftp" specify the location of a FTP server and the location of files on the servers. When a GRAM sees a FTP URL as the executable, the GRAM will contact the FTP server, download the executable from the sever, and then run the executable. The application itself downloads the parameter file specified in the arguments.

The last operator available in the RSL language is the multi-request operator, "+". This operator allows the user to specify several requests at once, all of which must be satisfied. This operator is currently used by the DUROC [13] coallocation library and is also useful to higher-level tools. The DUROC library provides support for running a single parallel application on more than one computer. For example, a user could run a simulation on 16 nodes of an Origin and a visualization on an ImmersaDesk using:

```
+(&(resourceManagerContact="denali.mcs.anl.gov:8713:/C=US/
      O=Globus/O=Argonne National Laboratory/OU=Mathematics
      and Computer Science Division/CN=denali.mcs.anl.gov-fork")
   (count=16)
   (directory="/sandbox/wsmith/cactus-3.2/cactus\_irix6")
   (arguments="/home/wsmith/cactus/neutron-128-128-yukon.par"))
 (&(resourceManagerContact="yukon.mcs.anl.gov:8713:/C=US/
      O=Globus/O=Argonne National Laboratory/OU=Mathematics
      and Computer Science Division/CN=denali.mcs.anl.gov-fork")
   (count=1)
   (directory="/sandbox/wsmith/cactus-3.2/viewer")
   (arguments="-host denali.mcs.anl.gov")))
```

Table 8.1 contains the main attributes users specify to use existing Globus resource management components, the operators that are used with the attributes in the current Globus tools, what type of data is expected as the values for the attributes, the default values for the attributes if no value is specified, and what the attributes describe.

Table 8.1: Available RSL attributes.

Resource Monitor Name	Compatible Operators	Data Type	Default Value	Description
arguments	=	String	NULL	Arguments to the executable
count	=	Integer	1	The number of processes to start
directory	=	String	user's home directory	The directory to uses as the working directory
executable	=	String	a.out	Executable to run
environment	=	String	NULL	Environment variables to set
jobtype	=	String	single	single, multiple, or mpi
stdin	=	String	/dev/null	Where to attach stdin of the processes to
stdout	=	String	/dev/null	Where to send stdout of the processes to
stderr	=	String	/dev/null	Where to send stdout of the processes to
queue	=	String	scheduler's default	The queue to submit the job to
project	=	String	user's default	The project to charge the computer time to

The following is a modified BNF grammar for the Resource Specification Language. Terminals appear in single quotes, eg 'terminal'. Lexical rules are provided for the implicit concatenation sequences in the form of conventional regular expressions; for the "implicit-concat" non-terminal rules, whitespace is not allowed between juxtaposed non-terminals. Grammar comments are provided in square brackets in a column to the right of the productions, eg [comment] to help relate productions in the grammar to the terminology used in the above discussion.

```
specification
  => relation
  => '+' spec-list [multi-request]
  => '&' spec-list [conjunct-request]
  => '|' spec-list  [disjunct-request]

spec-list
  => '(' specification ')' spec-list
  => '(' specification ')'

relation
  => 'variables' = binding sequence [variable def'ns]
  => attribute op value-sequence [relation]

binding-sequence
  => binding binding-sequence
  => binding
```

```
binding
  => '(' string-literal simple-value ')' [variable def'n]

attribute
  => string-literal  [attribute]

op
  => '=' |  '!=' | '>' | '>=' | '<' | '<='

value-sequence
  => value value-sequence
  => value

value
  => '(' value-sequence ')'
  => simple-value

simple-value
  => string-literal
  => simple-value '#' simple-value [concatenation]
  => implicit-concat
  => variable-reference
```

```
variable reference
  => '$(' string-literal ')' [variable ref.]
  => '$(' string-literal simple-value ')' [ref. w/ default]

implicit-concat
  => (unquoted-literal)? (implicit-concat-core)+  [implicit concat.]

implicit-concat-core
  => variable-reference
  => (variable-reference)(unquoted-literal)

string-literal
  => quoted-literal
  => unquoted-literal

quoted-literal
  => ' ' ' (([^'])|(' '' '))* ' ' '
  => ' " ' (([^"])|(' "" '))* ' " '
  => '^' c(([^c])|(cc))* c [user delimiter]

unquoted-literal
  => ([^ \t\v\n+&|()=<>!"'^#$])+ [nonspecial chars]

comment
  ==> '(*' (([^*])|('*'[^)]))* '*)' [comment]
```

8.4 Application Programming Interface

GRAM uses standard Globus module activation and deactivation. Before any GRAM client functions are called, the following function must be called:

```
globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE)
```

This function returns GLOBUS_SUCCESS if the GRAM client was successfully initialized, and the application is therefore allowed to subsequently call GRAM client functions. Otherwise, and error code is returned, and GRAM client functions should not be subsequently called. This function may be called multiple times.

To deactivate GRAM client, the following function must be called:

```
globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE)
```

This function should be called once for each time the GRAM client was activated.

```
int globus_gram_client_job_request(
    char *resource_manager_contact,
    char *description,
    const int job_state_mask,
    const char *callback_contact,
    char **job_contact)
```

Request access to interactive resources at the current time. A job request is atomic: either all of the requested processes are created, or none are created.

- `resource_manager_contact` - the contact information about the resource manager to which the request is submitted. This information is located in the MDS browser, or at http://www.mcs.anl.gov/bester/cgi-bin/gusto_status.cgi

- `description` - a RSL description of the requested job
- `job_state_mask` - 0, a bitwise OR of the GLOBUS_GRAM_CLIENT_JOB_STATE_* states, or GLOBUS_GRAM_CLIENT_JOB_STATE_ALL
- `callback_contact` - the URL which will receive all messages about the job
- `job_contact` - in a successful case, this is set to a unique identifier for each job.
- `globus_gram_client_job_request` returns GLOBUS_GRAM_CLIENT_SUCCESS if successful, or an error code.

```
int globus_gram_client_job_status(
    char *job_contact,
    int *job_status,
    int *failure_code)
```

This function returns the status of the job associated with the job contact.

- `job_contact` - the job_contact of the job in question.
- `job_status` - Set to one of GLOBUS_GRAM_CLIENT_JOB_STATE_* states when GLOBUS_SUCCESS is returned.
- `failure_code` - Set to an error code when GLOBUS_FAILURE is returned.
- `globus_gram_client_job_status` returns GLOBUS_SUCCESS or GLOBUS_FAILURE.

```
int globus_gram_client_job_callback_register(
```

```
    char *job_contact,
    const int job_state_mask,
    const char *callback_contact,
    int *job_status,
    int *failure_code)
```

This function registers the callback_contact for job state changes.

- `job_contact` - the job_contact of the job in question.
- `job_state_mask` - 0, a bitwise OR of the GLOBUS_GRAM_CLIENT_JOB_STATE_* states, or GLOBUS_GRAM_CLIENT_JOB_STATE_ALL
- `callback_contact` - the URL which will receive all messages about the job
- `job_status` - Set to one of GLOBUS_GRAM_CLIENT_JOB_STATE_* states when GLOBUS_SUCCESS is returned.
- `failure_code` - Set to an error code when GLOBUS_FAILURE is returned.
- `globus_gram_client_job_register` returns GLOBUS_SUCCESS or GLOBUS_FAILURE.

```
int globus_gram_client_job_callback_unregister(
    char *job_contact,
    const char *callback_contact,
    int *job_status,
    int *failure_code)
```

This function unregisters the callback_contact from future job state changes.

- `job_contact` - the job_contact of the job in question.
- `callback_contact` - the URL which will receive all messages about the job
- `job_status` - Set to one of GLOBUS_GRAM_CLIENT_JOB_STATE_* states when GLOBUS_SUCCESS is returned.
- `failure_code` - Set to an error code when GLOBUS_FAILURE is returned.
- `globus_gram_client_job_register` returns GLOBUS_SUCCESS or GLOBUS-_FAILURE.

```
int globus_gram_client_job_cancel(
    char *job_contact)
```

This function removes a PENDING job request, or kills all processes associated with an ACTIVE job, releasing any associated resources.

- `job_contact` - the job_contact of the job in question.
- `globus_gram_client_job_cancel` returns GLOBUS_GRAM_CLIENT_SUC--CESS if it is successful, or an error code.

```
int globus_gram_client_job_check(
    char *resource_manager_contact,
    const char *description,
    const float conf_percentage,
    globus_gram_client_time_t *estimate,
    globus_gram_client_time_t *interval\_size)
```

Note: This is not yet implemented

This function returns an estimate of the time it would take for a job of the description provided to reach an ACTIVE state.

- `resource_manager_contact` - the contact information about the resource manager to which the request is submitted. This information is located in the MDS browser, or at http://www.mcs.anl.gov/bester/cgi-bin/gusto_status.cgi
- `description` - a RSL description of the requested job
- `conf_percentage` - the required confidence level in the estimate. This user-specified number is taken by the resource manager which returns an interval. The RM has conf_percentage confidence that the start time will occur within this interval.

  For example, if conf_percentage = .9, the RM estimates that the start time will occur within the (estimate - interval_size/2, estimate + interval_size/2) with 90
- `estimate` - the estimated time in which the job will become ACTIVE.
- `interval_size` - the size of the confidence interval.
- `globus_gram_client_job_check` returns GLOBUS_GRAM_CLIENT_SUCCESS if it is successful, or an error code.

```
int globus_gram_client_job_start_time(
    char *job_contact,
    const float conf_percentage,
    globus_gram_client_time_t *estimate,
    globus_gram_client_time_t *interval\_size)
```

Note: This is not yet implemented

This function returns the estimated start time of a PENDING job or the actual start time of an ACTIVE job.

- `job_contact` - the job_contact of the job in question.
- `conf_percentage` - the required confidence level in the estimate. This user-specified number is taken by the resource manager which returns an interval. The RM has conf_percentage confidence that the start time will occur within this interval.

  For example, if conf_percentage = .9, the RM estimates that the start time will occur within the (estimate - interval_size/2, estimate + interval_size/2) with 90
- `estimate` - the estimated time in which the job will become ACTIVE.
- `interval_size` - the size of the confidence interval.
- `globus_gram_client_job_start_time` returns GLOBUS_GRAM_CLIENT_SUCCESS if it is successful, or an error code.

```
int globus_gram_client_job_contact_free(
    char *job_contact)
```

This function releases the resources storing the job contact.

- `job_contact` is the identifier returned with each job request.

```
int globus_gram_client_callback_allow(
    void callback_func(
```

```
        void * user_callback\_arg,
        char * job_contact,
        int state, int errorcode),
    void * user_callback_arg,
    char **callback_contact)
```

This function creates a TCP port on which it listens for any response from the Job Manager. These messages come in the form of GLOBUS_GRAM_CLIENT_SUCCESS, or any error message, indicating that the submission has not been successful.

- `callback_func` is a user-defined function with 4 parameters.
- `user_callback_arg` is any argument the user defines. (a general, all purpose parameter)
- `job_contact` is the contact of the specific job in question
- `state` returns any new job state code
- `errorcode` holds an error code code if state = FAILED
- `user_callback_arg` is any argument the user defines, and needs to pass into the callback_func
- `callback_contact` is the contact for the callback channel

`int globus_gram_client_callback_check()`

This function needs to be used to receive callbacks if GRAM monitors are not used.

8.5 Summary

In this chapter, we present a common interface to supercomputer scheduling systems. This interface consists of an application programming interface and a language to describe which resources are needed and how to run an application on those resources. We describe this interface in detail, including the available functions, how to use the functions, the scheduling model, and the resource specification language.

8.6 Future Work

There are several areas of future work. First, this interface is being extended to provide an interface to resources other than compute resources. The work in [25] slightly modifies this resource management interface to request network quality of service and percentages of other shared resources such as CPUs and disk I/O bandwidth. Second, this interface is being adopted as a standard in the metacomputing community and slight modifications may be necessary to support this effort.

Chapter 9

Conclusions

We address several problems in this work related to resource selection and scheduling in metacomputing systems. We present an information service that provides information about resources to users. This allows users to select the most appropriate resources for their applications. While this service is very useful, all of the information needed by users is not currently available. One such piece of information that is not currently available is when scheduling systems will start applications that have been submitted to them. This information would allow users to select computing resources based on when their application would start executing.

We investigate methods for predicting when schedulers will assign resources to applications. We propose and evaluate a general technique for maintaining a historical database and using this database to predict characteristics of data points. We use this technique for predicting application execution times and for predicting queue wait times. When we predict application run times, the applications are described by the characteristics that a user provides when they submit an application

to a parallel scheduler. We predict application run times using the run times of "similar" applications that have executed in the past. Initially, we do not know which characteristics to use to define which applications are similar and we do not know which of several statistical techniques to use to produce a prediction from similar applications. We search for which characteristics to use to define similar and how to produce a prediction using both greedy and genetic algorithm searches. We find that our prediction errors are between 29 and 54 percent of the mean run times of the four workloads we use to evaluate our technique and are significantly smaller than the errors of other run-time prediction techniques.

We use two techniques to predict queue wait times. The first technique uses run-time predictions and performs scheduling simulations of all the running and queued applications to predict when they will start executing. There are two disadvantages to this technique. First, exact knowledge of the scheduling algorithm is required and this knowledge can be difficult to determine about commercial scheduling systems. Second, this technique does not consider any applications that have not yet been submitted and with some scheduling algorithms, these applications can affect the start times of applications that are already in queues. We find that this technique has a prediction error of between 30 and 59 percent of the mean wait times. The second technique directly uses our prediction technique based on historical data. This technique characterizes the state of a scheduler and the application whose wait time is being predicted, finds similar scheduler/application states that have existed in the past, and then uses historical information of wait times in these similar states to produce a wait-time prediction. This technique has a prediction error of between 49 and 94 percent of mean wait times.

In addition to selecting which resources to use, users must be able to schedule

access to the resources. To assist in this, we helped to define a common interface to supercomputer scheduling systems. This interface allows a user to start, monitor, and terminate applications and can be layered atop various scheduling systems or run on computer systems without schedulers.

Further, many metacomputing applications require simultaneous access to resources and current scheduling systems do not provide this support when resources are controlled by more than one scheduler. To address this problem, we propose and evaluate techniques for reserving resources on supercomputers. We examine several different techniques for this and evaluate their performance based on changes in mean queue wait times and how near reservations are made to when the users initially request them. We find that if we cannot restart applications, we are forced to use maximum run times as predictions when scheduling. In this case, there is a 24 percent increase in wait time when 10 percent of the applications are reservations and a 144 percent increase in wait time when 20 percent of the applications are reservations. If we can restart applications, then we can use our run-time predictions when scheduling and the mean wait times decrease by 8 percent for the ANL workload and the mean difference between the requested reservation times and the actual reservation times decreases by 124 percent.

As a prelude to our work on techniques for reserving supercomputing resources, we improve the performance of scheduling algorithms by using more accurate run-time predictions. Several scheduling algorithms use run-time predictions and typically use the maximum run-times that are specified by the users. We find that using our run-time predictions instead of the user-specified maximum run-times decreases the average wait time by 7 percent and is only 5 percent worse on average than when actual run times are used as run time predictions.

Bibliography

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[2] Mirion Bearman. Tutorial on the ODP Trading Function. Technical report, University of Canberra, Australia, May 1996.

[3] W. Benger, I. Foster, J. Novotny, E. Seidel, J. Shalf, W. Smith, and P. Walker. Numerical Relativity in a Distributed Environment. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, April 1999.

[4] Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing '96*, 1996 (to appear).

[5] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Sixth Workshop on I/O in Parallel and Distributed Computer Systems*, May 1999.

[6] C. Mic Bownam, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. Harvest: A Scalable, Customizable Discovery and Access System. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado-Boulder, 1994.

[7] Allan Bricker, Michael Litzkow, and Miron Livny. *Condor Technical Summary.* Computer Sciences Department, University of Wisconsin - Madison, October 1991.

[8] S. Brunett, K. Czajkowski, S. Fitzgerald, I. Foster, A. Johnson, C. Kesselman, J. Leigh, and S. Tuecke. Application Experiences with the Globus Toolkit. In *7th IEEE Symposium on igh Performance Distributed Computing*, 1998.

[9] S. Brunett, D. Davis, T. Gottshalk, P. Messina, and C. Kesselman. Implementing Distributed Synthetic Forces Simulations in Metacomputing Environments. In *Proceedings of the Heterogeneous Computing Workshop*, March 1998.

[10] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. Technical Report CS-95-313, University of Tennessee, November 1995.

[11] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *The 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[13] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metasystems. *Lecture Notes on Computer Science*, 1998.

[14] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The Network-Enabled Optimization System (NEOS) Server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[15] Tom DeFanti, Ian Foster, Mike Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 1996 (to appear).

[16] Allen Downey. Predicting Queue Times on Space-Sharing Parallel Computers. In *International Parallel Processing Symposium*, 1997.

[17] N. R. Draper and H. Smith. *Applied Regression Analysis, 2nd Edition.* John Wiley and Sons, 1981.

[18] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, 1995.

[19] Dror Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790, IBM T.J. Watson Research Center, October 1995.

[20] Dror Feitelson and Bill Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. *Lecture Notes on Computer Science*, 949:337–360, 1995.

[21] Dror Feitelson and Ahuva Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, 1998.

[22] S. Fitzgerald, I. Foster, C. Kesselman, G. vol Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.

[23] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing Multiple Communication Methods in High-Performance Networked Computins Systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[24] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of the ACM/IEEE SC98 Conference*. ACM Press, 1998.

[25] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *International Workshop on Quality of Service*, 1999.

[26] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Communications Security*, pages 83–92, 1998.

[27] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[28] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, and Steven Tuecke. Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment. In *The 5th IEEE Symposium on High Performance Distributed Computing*, August 1996.

[29] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[30] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffmann, 1999.

[31] Richard Gibbons. A Historical Application Profiler for Use by Parallel Scheculers. *Lecture Notes on Computer Science*, 1297:58–75, 1997.

[32] Richard Gibbons. A Historical Profiler for Use by Parallel Schedulers. Master's thesis, University of Toronto, 1997.

[33] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[34] Andrew Grimshaw and more. Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.

[35] Andrew S. Grimshaw, Anh Nguyen-Tuong, and William. A. Wulf. Campus-Wide Computing: Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, 1995.

[36] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, University of Virginia, June 1994.

[37] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[38] S. Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. *Lecture Notes on Computer Science*, 1162:27–40, 1996.

[39] T. Howes. RFC 1558: A String Representation of LDAP Search Filters, 1993.

[40] Timothy A. Howes and Mark C. Smith. *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol.* Macmillan Technical Publishing, 1997.

[41] M. Iverson, F. Ozguner, and L. Potter. Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. In *Proceedings of the IPPS/SPDP'99 Heterogeneous Computing Workshop*, 1999.

[42] N. Kapadia, J. Fortes, and C. Brodley. Predictive Application Performance Modeling in a Computational Grid Environment. In *Proceedings of the 8th High Performance Distributed Computing Conference*, 1999.

[43] C. Lee, C. Kesselman, and S. Schwab. Near-real-time Satellite Image Processing: Metacomputing in CC++. *Computer Graphics and Applications*, 16(4):79–84, 1996.

[44] David A. Lifka. The ANL/IBM SP Scheduling System. *Lecture Notes on Computer Science*, 949:295–303, 1995.

[45] M. Litzkow, M. Livney, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[46] M. Litzkow and M. Livny. Experience With The Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, 1990.

[47] G. Mansfield, T. Johannsen, and M. Knopper. Charting Networks in the X.500 Directory. RFC 1609, 03/25 94. (Experimental).

[48] B. Clifford Neuman. Prospero: A Tool for Organizing Internet Resources. *Electronic Networking: Research, Applications, and Policy*, 2(1):30–37, Spring 1992.

[49] B. Clifford Neuman and Steven Seger Augart. Prospero: A Base for Building Information Infrastructure. In *Proceedings of INET'93*, 1993.

[50] B. Clifford Neumann and Santosh Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[51] Platform Computing Corporation. *LSF 2.2 User's Guide*, Febuary 1996.

[52] Platform Computing Corporation. *LSF Administrator's Guide*, Febuary 1996.

[53] Platform Computing Corporation. *LSF Programmer's Guide*, Febuary 1996.

[54] Jennifer Schopf and Francine Berman. Performance Prediction in Production Environments. In *14th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing*, 1998.

[55] Warren Smith, Ian Foster, and Valerie Taylor. Predicting Application Run Times Using Historical Information. *Lecture Notes on Computer Science*, 1459:122–142, 1998.

[56] Warren Smith, Valerie Taylor, and Ian Foster. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *Proceedings of the IPPS/SPDP'99 Workshop on Job Scheduling Strategies for Parallel Processing*, 1999.

[57] Parallel Batch System. http://pbs.mrj.com.

[58] G. von Laszewski, M. Su, J. Insley, I. Foster, J. Bresnahan, C. Kesselman, M. Thiebaux, M. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-Time Analysis, Visualization, and Steering of Microtomography Experiments at Photon Sources. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, April 1999.

[59] Gregor von Laszewski. *A Parallel Data Assimilation System and Its Implications on a Metacomputing Environment.* PhD thesis, Syracuse University, Dec 1996.

[60] C. Weider, J. Fullton, and S. Spero. RFC 1913: Architecture of the Whois++ Index Service. Technical report, Network Working Group, Febuary 1996.

[61] Neil Weiss and Matthew Hassett. *Introductory Statistics.* Addison-Wesley, 1982.

[62] Richard Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. Technical Report TR-CS96-494, U.C. San Diego, October 1996.

[63] W. Yeong, T. Howes, and S. Kille. RFC 1777: Lightweight Directory Access Protocol. Technical report, Network Working Group, 1995.

[64] D. Zotkin and P. Keleher. Job-Length Estimateion and Performance in Backfilling Schedulers. In *Proceedings of the 8th Conference on High Performance Distributed Computing*, pages 236–243, 1999.

Appendix A

Statistical Methods

We use statistical methods [61, 17] to calculate run-time estimates and confidence intervals from categories. A category contains a set of data points called a *sample*, which are a subset of all data points that will be placed in the category, the *population*. We use a sample to produce an estimate using either a mean or a linear regression. This estimate includes a confidence interval that is useful as a measure of the expected accuracy of this prediction. If the $X\%$ confidence interval is of size c, a new data point will be within c units of the prediction $X\%$ of the time. A smaller confidence interval indicates a more accurate prediction.

A mean is simply the sum of the data points divided by the number of data points. A confidence interval is computed for a mean by assuming that the data points in our sample S are an accurate representation of all data points in the population P of data points that will ever be placed in a category. The sample is an accurate representation if they are taken randomly from the population and the sample is large enough. We assume that the sample is random, even though it

consists of the run times of a series of applications that have completed in the recent past. If the sample is not large enough, the sample mean $\overline{x}$ will not be nearly equal to the population mean μ, and the sample standard deviation s will not be near to the population standard deviation σ. The prediction and confidence interval we compute will not be accurate in this case. In fact, the central limit theorem states that a sample size of at least 30 is needed for $\overline{x}$ to approximate μ, although the exact sample size needed is dependent on σ and the standard deviation desired for $\overline{x}$ [61].

We used a minimum sample size of 2 when making our predictions in practice. This is because while a small sample size may result in $\overline{x}$ not being nearly equal to μ, we find that an estimate from a category that uses many characteristics but has a small sample is more accurate than an estimate from a category that uses few characteristics but has a larger sample size.

The $X\%$ confidence interval can be computed when using the sample mean as a predictor by applying Chebychev's theorem. This theorem states that the portion of data that lies within k standard deviations to either side of the mean is at least $1 - \frac{1}{k^2}$ for any data set. We need only compute the sample standard deviation and k such that $1 - \frac{1}{k^2} = \frac{X}{100}$.

Our second technique for producing a prediction is to perform a linear regression to a sample using the equation

$$t = b_0 + b_1 n,$$

where n is the number of nodes requested and t is the run time. This type of prediction attempts to use information about the number of nodes requested. A confidence interval can be constructed by observing how close the data points are to this line. The confidence interval is computed by the equation

$$t_{\frac{\alpha}{2}}\sqrt{MSE}\sqrt{1+\frac{1}{N}+\frac{(n_0-\bar{n})^2}{\sum n^2-\frac{\left(\sum n\right)^2}{N}}},$$

where N is the sample size, MSE is the mean squared error of the sample, n_0 is the number of nodes requested for the application being predicted, and $\bar{n}$ is the mean number of nodes in the sample. Alpha is computed with the equation

$$\alpha = 1 - \frac{X\%}{100}$$

if the $X\%$ confidence interval is desired and $t_{\frac{\alpha}{2}}$ is the Student's t-distribution with $N-2$ degrees of freedom [61, 17].